# Dokumentation: Programmierung in Kryptowährungen

Harald Heckmann

harald.b.heckmann@student.hs-rm.de

14. September 2018

# Inhalt

# 1. Einleitung

Satoshi Nakamoto ist der Name oder ein Pseudonym des Autors des Papers zur ersten Kryptowährung „Bitcoin“ [Nak09a] aus dem Jahr 2009. Das Paper wurde von ihm kurz nach der Finanzkrise von 2008 veröffentlicht und kann als Reaktion darauf betrachtet werden. Die Intention war, eine von der Regierung und Banken abgekoppelte digitale Währung zu erschaffen, über die kein Individuum alleinig die Kontrolle hat. Somit kam nur ein dezentrales System in Frage. Alle Teilnehmer sind in dem Bitcoin-Netzwerk gleichberechtigt, niemand hat besondere Privilegien und alle handeln nach den gleichen Regeln.
Bitcoin ist transparent, jeder kann zu einem beliebigen Zeitpunkt aktuelle Transaktionsaufträge und die gesamte Transaktionshistorie nachvollziehen, welche in der Blockchain (die Buchhaltung) festgehalten wird. Weiterhin kann man sich darauf verlassen, dass die Blockchain nicht nachträglich verfälscht wurde, denn sie ist (praktisch) unveränderbar. Zusätzlich funktioniert das Netzwerk ohne Vertrauen, denn das verwendete Protokoll ist offen einsehbar. Letztendlich ist jeder Netzwerkknoten skeptisch gegenüber jedem weiteren Netzwerkknoten. Diese prüfen nämlich, ob alle Transaktionsaufträge gültig sind und ob die Blockchain fehlerfrei ist. Ursprünglich bestand die Notwendigkeit der Programmierung innerhalb einer Kryptowährung darin, Transaktionen zu authorisieren und diese anschließend validieren zu können. Wie die dazu verwendete Sprache aussieht und warum diese und das dazugehörige Konzept zur Programmierung innerhalb der Kryptowährung nicht für alle Szenarien ausreichend waren, wird im Laufe dieses Dokuments erläutert.

## 1.1. Guthaben Aufbewahrung und Ermittlung - Konzepte

Die Aufbewahrung, der Transfer und die Ermittlung von Guthaben unterscheiden sich zwischen vielen Kryptowährungen von Grund auf. Bitcoin, die erste Kryptowährung, setzte auf ein Konzept in dem es kein explizites Guthaben gibt, sondern dieses nur implizit vorliegt. Das Guthaben ergibt sich aus der Menge der Transaktionen, die einer Adresse (zumeist der Hash eines öffentlichen Schlüssel) zugeordnet sind. Das Guthaben ist somit als Information verborgen. Streng genommen existieren Konten als Datenstruktur nicht in solchen Kryptowährungen, sondern sind eine weitere Abstraktionsstufe aus gesammelten Informationen.
Neue Kryptowährungen wählen teilweise einen anderen Weg. In diesen existiert ein Konto als Datenstruktur, worin das Guthaben explizit aufbewahrt wird. Wesentlich müheloser geht die Ermittlung des Guthaben in solchen Kryptowährungen von statten, denn hierbei reicht eine einfache Anfrage nach dem Guthaben an das Konto.
Beide Konzepte haben unterschiedliche Auswirkungen auf die Verwendung einer Programmiersprache in der Kryptowährung zur Folge. Im folgenden werden die beiden Konzepte genauer beschrieben, sodass anschließend auf die Programmierung in Kryptowährungen und letztendlich auf ein Vergleich dieser genauer eingegangen werden kann.

### 1.1.1. Implizite Daten

Wie bereits erwähnt verwendet Bitcoin ein Protokoll, indem es keine Konten gibt. Diese sind letztendlich abstrakte Konstrukte, welche die Informationen aus der Kryptowährung für Menschen lesbar und einfach verständlich darstellen. Woher kommt nun die Information, wie viel Guthaben jemand besitzt? Im Prinzip ist es ganz einfach. Es existiert eine Menge in Hauptspeicher eines jeden vollwertigen Netzwerkknotens, in der die Einträge jeweils aus einem Guthaben und einem kryptographischen Puzzle bestehen. Diese Menge wird „ Unspent Transaction Output (UTXO)-Menge“ genannt. Das gesamte Guthaben ergibt sich aus dem Guthaben aller Einträge, zu denen man die Lösung des kryptographischen Puzzles kennt, wie in Abbildung 1.1 vereinfacht dargestellt wird. [Nak09a; Ant15]

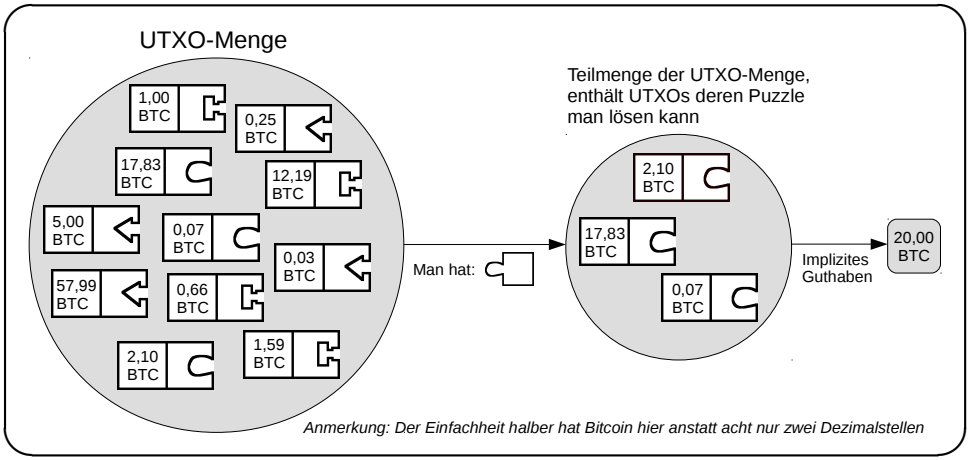


Abb. 1.1.: Das UTXO-Set und das implizite Guthaben

Eine erfolgreiche Transaktion führt zu einem Zustandswechsel der UTXO-Menge. Eine gültige und akzeptierte Transaktion bezweckt, dass mindestens ein UTXO konsumiert (Eingabe) und mindestens ein UTXO produziert (Ausgabe) wird. Das kann dazu führen, das implizit Guthaben den Besitzer wechselt.

**Beispiel: Guthaben transferieren in Bitcoin**
Soll nun Guthaben von einem Besitzer A zu einem anderen Besitzer B übertragen werden, so erzeugt Besitzer A eine Transaktion. Diese Transaktion enthält als Eingaben UTXO-Einträge, deren kryptographisches Puzzle Besitzer A lösen kann. Die Einträge werden so gewählt, dass die Summe der darin enthaltenen Guthaben etwas mehr als dem zu übertragenden Guthaben ergibt. Die Ausgaben sind in zwei Teile geteilt:

1. UTXO mit Guthaben, dass an Besitzer B übertragen werden soll. Enthält ein kryptographisches Puzzle, das Besitzer B lösen kann.

2. UTXO mit Guthaben (Wechselgeld), dass zurück an Besitzer A übertragen wird. Enthält ein kryptographisches Puzzle, das Besitzer A lösen kann.

Wichtig ist zu beachten, dass die Summe aus den beiden erzeugten UTXO geringer ist als das Guthaben aus den Eingaben. Die Differenz erhält derjenige, der einen gültigen Block (Buchhaltungsseite) erzeugt, in der die Transaktion enthalten ist. Diese Differenz wird Transaktionsgebühr genannt. Abbildung 1.2 stellt diese Transaktion grafisch dar.

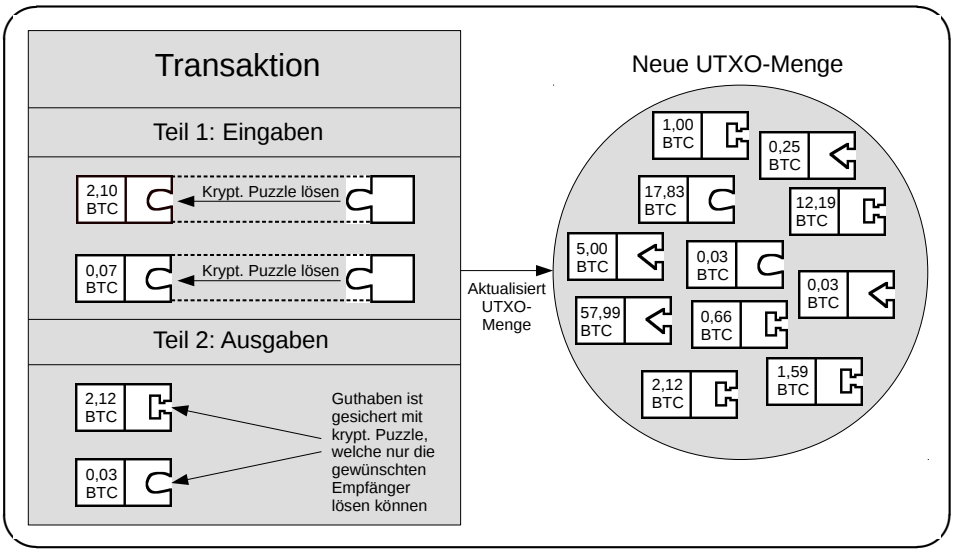


Abb. 1.2.: Eine Transaktion in Bitcoin

### 1.1.2. Explizite Daten

Anders als in Bitcoin sind Konten in Ethereum keine abstrakten Konstrukte, sondern sie existieren als Datenstrukturen im Hauptspeicher eines jeden (vollwertigen) Knoten des Netzwerks. Die Daten eines Kontos sind hierbei fest verknüpft mit der dem Konto zugrunde liegenden Adresse. Die Sammlung aller Abbildungen zwischen Adresse und Kontodaten wird „World State“ (Weltzustand) genannt. Wie der Name schon suggeriert ist hier der gesamte Zustand des Ethereum-Universum abgebildet. Der World State ist, zusammen mit zwei weiteren Datensätzen, in der sogenannten „State Database (DB) “ (Zustandsdatenbank) abgespeichert [But13; Woo14]. Abbildung 1.3 stellt die

State DB dar.

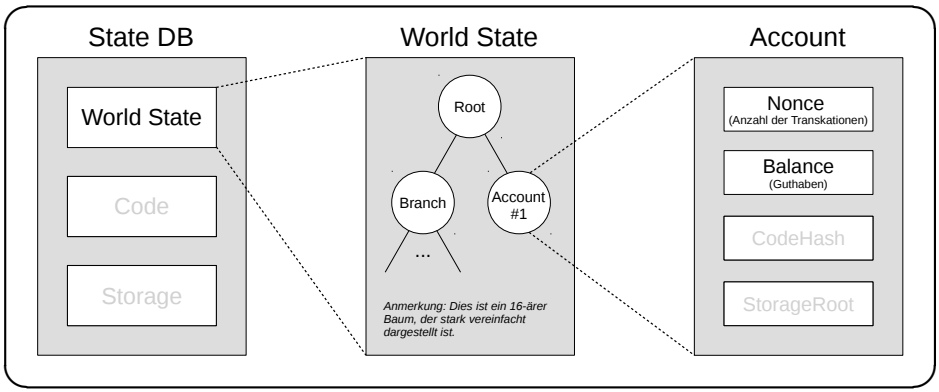


Abb. 1.3.: State DB

Abbildung 1.3 zeigt unter Anderem, dass das Guthaben als Zustand in dem Datensatz des jeweiligen Kontos vorhanden ist. Die Daten bezüglich eines Kontos sind somit explizit vorhanden. Weiterhin sind zwei weitere Datensätze, Code und Storage zu sehen. Das liegt daran, dass es zwei Arten von Konten in Ethereum gibt: Standardkonten und Programmkonten. In diesem Kapitel werden zunächst nur Standardkonten und Transaktionen zwischen diesen betrachtet.
Eine erfolgreiche Transaktion führt in einer Kryptowährung wie Ethereum dazu, dass der Datensatz des Kontos sich ändert. Diese bezweckt somit letztendlich eine Zustandsänderung der State DB. Die Transaktionen entsprechen, anders als Transaktionen in Kryptowährungen mit impliziten Daten, von Form und Auswirkung eher den Transaktionen aus der Bankenwelt. Diese enthalten den Empfänger, den Betrag, eine digitale Signatur und noch einige zusätzliche Angaben. Wenn eine Transaktion in den nächsten Block aufgenommen wird, kann jeder öffentlich diese Transaktion einsehen und anschließend die entsprechenden Änderungen in seiner State DB vornehmen, sodass diese aktuell ist. Es existieren Daten im Block, die mit geeigneten Mechanismen zusammen einen Abgleich des lokalen World State ermöglichen, sodass sichergestellt werden kann das der lokale World State korrekt ist.

## 1.2. Programmierung in Kryptowährungen

Schon seit der Veröffentlichung des ersten Bitcoin Clients v0.1.0 [Nak], welcher von Satoshi Nakamoto im Jahr 2009 über eine Kryptographie Mailingliste [Nak09b] verteilt wurde, ist die Programmierung innerhalb von Bitcoin möglich. Somit ist die Programmierung in Kryptowährungen schon seit Beginn dieser vorgesehen. Allerdings haben sich die Anforderungen im Laufe der Zeit erweitert, und zwar so sehr, dass die Ursprünglichen Konzepte nicht mehr genügten und neue hervorgebracht werden mussten, um den Anforderungen gerecht zu werden.

### 1.2.1. Ursprüngliche Notwendigkeit

Was war die ursprüngliche Notwendigkeit der Programmierung in einer Kryptowährung? Bitcoin verwendet eine Skriptsprache um die kryptographischen Puzzle abzubilden, welche einen UTXO vor der Ausgabe schützen. Weiterhin dient die Skriptsprache der formulierung der Lösung des kryptographischen Puzzle. Die Puzzleteile aus Abbildung 1.2 sind somit jeweils ein Skript. Diese Skripts bieten einige Möglichkeiten Aufgaben und Lösungen zu formulieren, allerdings sind die Möglichkeiten stark eingeschränkt, da die Skriptsprache viele Sprachelemente nicht enthält die für ein Turing-Vollständiges Programm notwendig sind. Es folgen zwei textuelle Beispiele, die zur Veranschaulichung der Verwendung der Skriptsprache dienen. Das erste Beispiel zeigt das Standardverfahren, welches verwendet wird um den Empfänger mittels seiner Adresse festzulegen. Das zweite Beispiel zeigt ein imaginäres Verfahren, welches eine Belohnung für denjenigen bereitstellt, der das knacken eines in Bitcoin verwendeten Hashalgorithmus bewerkstelligt.

**Beispiel 1: Guthaben an Besitzer einer Adresse binden**
Aufgabe: Gegeben einer Adresse $x$ und einer Transaktion $t$, in der dieses Skript enthalten ist, erzeuge eine gültige digitale Signatur $d_{xt}$ (wer hat den privaten Schlüssel zu der Adresse $x$?).
Lösung: Digitale Signatur $d_{xt}$ und öffentlicher Schlüssel zum verifizieren der Signatur.

**Beispiel 2: Belohnung für das knacken eines in Bitcoin verwendeten Hashalgorithmus**

Aufgabe: Gegeben drei Hashes, finde die Rohdaten dazu.
Lösung: Rohdaten aller drei Hashes.

### 1.2.2. Steigender Bedarf, mangelnde Möglichkeiten

Die Skriptsprache in Bitcoin ermöglicht zwar viele Puzzles zu erzeugen, allerdings eignet sie sich nicht dazu alle Abläufe zu implementieren. Mangelnde essentielle Operationen, die fehlende Möglichkeit der Skripte Daten persistent zu speichern, abzurufen und zu verändern sowie die Einschränkung Benutzereingaben nicht verarbeiten zu können, machen die Skriptsprache in Bitcoin für die Implementierung vieler Anwendungsfälle ungeeignet. Dies wird genauer in Kapitel 2.1 beschrieben.
Entwickler erkannten diesen Bedarf und veröffentlichten 2015 Ethereum, die erste Kryptowährung die eine Turing-Vollständige Programmiersprache enthält. Die Programme werden in einer Virtual Machine (VM), der Ethereum Virtual Machine (EVM), interpretiert. Die EVM läuft auf allen gängigen Prozessorarchitekturen, sodass jeder Netzwerkknoten in der Lage ist den Code auszuführen. Mit der Veröffentlichung von Ethereum wurde der erste dezentrale Computer erschaffen, dessen Zustandswechsel Weltweit abgeglichen und persistent sowie unveränderbar (in einer Blockchain) abgespeichert werden.
Dieses Konzept gibt Entwicklern sehr viele Freiheiten bezüglich der Programmierung innerhalb einer Kryptowährung. Diese vielen Freiheiten führen allerdings auch zu Gefahren, denn es ist unheimlich schwer vorher abzuwägen, welche neuen Angriffsszenarien auf die Kryptowährung dadurch entstehen. Mögliche Gefahren werden in Kapitel 3.3 betrachtet.

### 1.2.3. Smart Contracts

Von der Grundidee her ist ein Smart Contract die digitale Version eines physischen Vertrages. Dabei werden Klauseln des Vertrages, wie z.B. „Wenn Ereignis A eintritt, überweise Guthaben B and Teilnehmer C“, in Code abgebildet. Die Vorteile hierbei sind, dass die Verträge unveränderlich in der Blockchain festgehalten werden und ein Betrug bei ordentlich durchdachten und Entwickelten Verträgen beinahe ausgeschlossen ist. Ein weiterer Vorteil gegenüber physischen Verträgen ist der, das viele Prozesse automatisiert wer-

den können. So muss ein Teilnehmer nicht prüfen ob der Vertrag erfüllt wurde und dann agieren, sondern das geschieht ohne Einwirkung des Teilnehmers. Wenn es gewünscht ist kann ein Vertrag dynamisch angepasst werden, sodass Beispielsweise das nachträgliche hinzufügen von Teilnehmer möglich ist. Dies muss allerdings bei der Programmierung dessen berücksichtigt werden, denn ein Vertrag (der Code) der in der Blockchain festgehalten wird ist unveränderlich. Smart Contracts haben weiterhin den Vorteil, das diese mit weiteren Smart Contracts interagieren können. So können sehr komplexe Netzwerke aus Verträgen entstehen. Die Abläufe im Vertrag sind strengt kontrolliert, denn jeder Netzwerkknoten validiert den Ablauf des Vertrages. Die Ziele von Smart Contracts sind zusammenfassend das erreichen einer höheren Vertragssicherheit, die Automatisierung von Vertragsabläufen sowie eine Reduktion der Transaktionskosten.

# 2. Bitcoin Scripting Language - Nicht Turing-Vollständig

Die Bitcoin Scripting Language (im folgenden nur „Script“ gennant) ist eine Stackbasierte Sprache mit starken Einschränkungen. Wie aus der Einleitung bereits hervorgeht, dient sie dazu UTXOs vor der Ausgabe durch Unbefugte zu schützen. Der UTXO kann ausgegeben werden wenn die Ausführung vom Skript, das die Lösung zum kryptographischen Puzzle präsentiert und anschließend die Ausführung vom Skript, welches das kryptographische Puzzle erzeugt, nicht FALSE auf dem Stack hinterlässt oder ein Operator explizit die Ausführung der Skripts unterbricht. Im folgenden werden die Einschränkungen dieser Sprache beschrieben sowie einige Beispiele kryptographischer Puzzles in dieser Sprache erstellt und erklärt.

## 2.1. Einschränkungen

Wie bereits in dem Titel dieses Kapitels erwähnt wird, ist Script nicht Turing-Vollständig. Doch ist das schon alles an Einschränkungen oder gibt es noch mehr? Tatsächlich gibt es noch mehr Einschränkungen, auf die im folgenden genauer eingegangen wird.

### 2.1.1. Sprachelemente

Um Script möglichst sicher zu gestalten wurden beim dem Design dieser Sprache absichtlich einige, für ein Turing-Vollständiges Programm essentielle, Operationen weggelassen. Eine Liste aller Sprachelemente ist online verfügbar [Coma]. Anschließend wird eine Liste präsentiert, welche die Möglichkeiten zusammenfassend aufzeigt.

### Flusskontrolle

- Wenn, Wenn nicht, Andernfalls

### Bitweise Operatoren

- Gleichheit

### Arithmetische Operatoren

- Addition und Subtraktion
- Negation und absoluter Wert
- Gleichheit, Größer, Größergleich, Kleiner, Kleinergleich
- Min, Max, Intervall

### Textbearbeitung

- Länge des Textes

### Handhabung des Stacks

- Ist vollständig, d.H. hinzufügen, entfernen, Reihenfolge beliebig ändern.

### Besonderes

- Benötigte Kryptographie (Hashing, digitale Signaturen)
- absolute und relative Zeitanforderungen.

Was fehlt in dieser Liste? Erstens fehlen Schleifen. Das ist eine der wichtigsten Operationen um komplexe Abläufe darzustellen. Dadurch das Schleifen fehlen, fällt gleich ein Großteil der Möglichkeiten weg. Was dadurch gewonnen wird ist, dass die Programme eine statische Laufzeit haben und leicht analysiert werden können. Bedenkt man nun das jeder (vollwertige) Netzwerkknoten in Bitcoin jede Transaktion validiert, das heißt die Skripts darin ausführt und das Ergebnis prüft, stellt man fest das Endlosschleifen oder ähnliches in den Skriptsn die Dienste des Netzwerks außer Gefecht setzten könnten. Somit kann das verbieten von Schleifen als Austausch zwischen Möglichkeiten und Sicherheit gesehen werden. Letzteres gilt als Argument für alle verbotenen Operatoren.
Zweitens fehlen wichtige bitweise Operatoren. Diese fehlen bis auf Gleichheit vollständig. Ohne Schleifen ist es auch nicht möglich diese mittels Arithmetischen Operatoren zu rekonstruieren. Damit sind Modifikationen von einzelnen Bits nicht gezielt möglich.
Wichtige Arithmetische Operatoren wie die Multiplikation, die Division, Modulo und Shift sind ebenfalls nicht möglich und können mit Script nicht konstruiert werden. Komplexere Berechnungen sind somit ausgeschlossen.

### 2.1.2. Interaktion mit Script

in Script beschriebene Skripts dienen, wie bereits bekannt ist, nur dem Zweck ein UTXO vor unbefugter Verwendung zu schützen. Der einzige Zeitpunkt an dem die Ausführung dieser Skripts (Puzzle und dessen Lösung) stattfindet ist dann, wenn eine Transaktion validiert wird. Die Ausführung der Skripts führt bei Erfolg dazu, dass der Zustand der UTXO-Menge sich ändert und die Skripts konsumiert werden. Die einzige Möglichkeit mit einem Skript zu interagieren hat ein Benutzer also nur, wenn er eine Transaktion an das Netzwerk propagiert, die bezweckt das jenes Skript konsumiert wird. Die einzige erfolgreiche Interaktion mit dem Skript zerstört dieses somit, was bedeutet das weitere Interaktion mit diesem nicht mehr möglich sind. Interaktionen mit solchen Skripts die den Zustand dieser ändern oder deren Zustand abrufen war nie als Möglichkeit vorgesehen.

### 2.1.3. Persistente Daten

In Kapitel 2.1.1 wurden keine Operatoren zum persistenten Speichern sowie modifizieren dieser angegeben. Es gibt allerdings einen Tricks, mit dem man zumindest Daten speichern kann. Wenn eine Transaktion in den nächsten Block hinzugefügt wird und somit letztendlich in der Blockchain landet, bleibt sie dort für immer unveränderbar und ist öffentlich einsehbar. In den Transaktionen stehen die Puzzles und Lösungen, die in Script verfasst wurden. Das Bedeutet, wenn es möglich ist in das Skript selbst Daten zu verpacken, dann werden diese Daten ebenfalls persistent und unveränderbar in der Blockchain abgelegt. Das ist in der Tat möglich, allerdings haben sich die Verfahren dazu etwas geändert.
Damals wurden Daten, zumeist Strings oder Hashes, in den Skripten (dem Puzzle, nicht der Lösung dessen) oftmals so abgelegt, dass es zu diesen Skripten kein passendes Gegenstück gab. Das bedeutet, dass der UTXO in dem diese enthalten waren niemals ausgegeben werden konnte. Meistens geschah dies aus Unwissenheit bezüglich der Konsequenzen oder weil dem Ersteller des Puzzles keine alternative Lösung bekannt war. UTXOs liegen, anders als die Blockchain, im Hauptspeicher eines (vollwertigen) Bitcoin Netzwerkknotens. Wenn diese nun nicht ausgegeben werden können, „verstopfen“ sie den im Vergleich zu Festplatten relativ teuren und sehr begrenzten Hauptspeicher. Die Entwickler von Bitcoin einigten sich auf einen Kompromiss. Dieser beinhaltet die Einführung eines neuen Operators, OP_RETURN, welcher es ermöglicht maximal 80 Byte Daten pro UTXO hinzuzufügen und maximal einen pro Transaktion zu erzeugen. Einige sehen das Speichern willkürlicher Daten in der Blockchain von Bitcoin als Missbrauch dieser, da sie dadurch zweckentfremdet sowie zu schnell an Größe gewinnen würde. Tatsächlich kamen dabei jedoch schon einige nützlichen Anwendungen zustande, die den intrinsischen Wert von Bitcoin steigern. In einer Untersuchung des OP_RETURN Operators stellte sich heraus [PB17], dass nur 0,3% der Daten in der Blockchain mittels OP_RETURN hinzugefügt wurden. Ein Kompromiss ist es deshalb, da das Speichern von Daten in der Blockchain nicht verhindert werden kann und stattdessen ein Standard erzeugt wird der dies möglichst schonend ermöglicht.
Durch einige Tricks ist es also möglich, Daten persistent zu speichern. Script ist jedoch nicht in der Lage die Daten aus der Blockchain wieder auszulesen, sodass eine Weiterverarbeitung dieser nicht vom Skript selbstständig bewältigt

werden kann. Wäre Script in der Lage die Daten wieder auszulesen, so würde für jede Modifikation eine weitere Kopie in der Blockchain abgelegt werden, da die ursprünglichen Daten in der Blockchain nicht verändert werden können. Das heißt, alle Zustände die ein Wert jemals hatte wären in der Blockchain enthalten, was diese vermutlich ziemlich schnell künstlich aufblasen würde.

## 2.2. Skripts

Der folgende Abschnitt beschäftigt sich mit simplen Skripts, die in Bitcoin als kryptographisches Puzzle und deren Lösung fungieren. In diesem Teil werden der Code der Skripts sowie deren Ablauf genauer beschrieben. Das erste Beispiel zeigt zwei Skripts, die ein klassisches kryptographisches Puzzle und deren Lösung darstellen. Diese sind Standardskripts, welche für normale Transaktionen verwendet werden. Dieses Verfahren bezeichnet man als „ Pay-to-Public-Key-Hash (P2PKH)“. Das zweite Beispiel zeigt, wie die Ausgabe eines UTXO die Angabe mehrere digitalen Signaturen erfordert, welche von bestimmten Adresse abstammen müssen. Das dritte Beispiel zeigt wie ein Belohnungs-Rätsel erzeugt wird, das durch die Angabe der Rohdaten bezüglich dreier Hashes gelöst wird. Das letzte Beispiel zeigt, wie Daten nach dem Standardverfahren in der Blockchain gespeichert werden. Es wird immer zunächst die Lösung und anschließend die Aufgabe ausgeführt.

### 2.2.1. 1. Beispiel: P2PKH Skript

Ziel: UTXO erfordert eine gültige digitale Signatur des Besitzers
Aufgabe: OP_DUP OP_HASH160 OP_20 <Adresse$_x$>
OP_EQUALVERIFY OP_CHECKSIG
Lösung: OP_72 <Signatur$_{xt}$> OP_33 <PubKey$_x$>

P2PKH Skripts benötigen zu einer gegebenen Adresse $Adresse_x$, die aus einem öffentlichen Schlüssel $PubKey_x$ abgeleitet wird, eine gültige digitale Signatur für Teile der Transaktion $t$ in der diese Skripts enthalten sind.
Zuerst wird das Lösungsskript ausgeführt. Dieses macht folgendes: Es legt eine digitale Signatur und einen öffentlichen Schlüssel, mit dem die digitale Signatur überprüft werden kann, ab. Anschließend wird das Aufgabenskript ausgeführt, um die gegebene Lösung zu überprüfen. Das Aufgabenskript übernimmt

den Stack, der die digitale Signatur und den öffentlich Schlüssel enthält. Anschließend erzeugt es eine Adresse aus dem öffentlichen Schlüssel, der auf dem Stack liegt. Das geschieht durch hashing mittels SHA256 wessen Ergebnis mit RIPEMD160 gehasht wird. Danach wird die aus dem öffentlichen Schlüssel erzeugte Adresse mit der Adresse abgeglichen, die im Ausgabeskript festgehalten ist und die Adresse angibt, die den UTXO ausgeben darf. Wenn der Vergleich misslingt bricht die Ausführung ab und die Ausgabe des UTXO wird verweigert. Der Vergleich der Adresse dient dazu zu prüfen, ob der öffentliche Schlüssel der zum verifizieren der digitalen Signatur verwendet wird nicht irgendein beliebiger ist, sondern tatsächlich der Schlüssel aus der die Adresse erzeugt wird die den UTXO ausgeben darf. Bei Erfolg wird abschließend die digitale Signatur mit dem öffentlichen Schlüssel verifiziert. Wenn das gelingt, darf der UTXO ausgegeben werden. Abbildung 2.1 stellt den Ablauf des Lösungsskripts im Erfolgsfall und Abbildung 2.2 den Ablauf des Aufgabenskripts im Erfolgsfall dar.

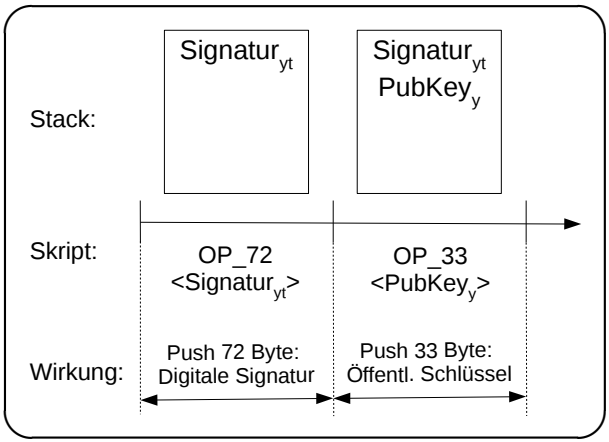


Abb. 2.1.: P2PKH Lösungsskript

### 2.2.2. 2. Beispiel: Multisignature Skript

Ziel: UTXO erfordert gültige Signaturen mehrerer Besitzer
Aufgabe: OP_M <PubKey$_{x1}$> <...> <PubKey$_{xN}$>

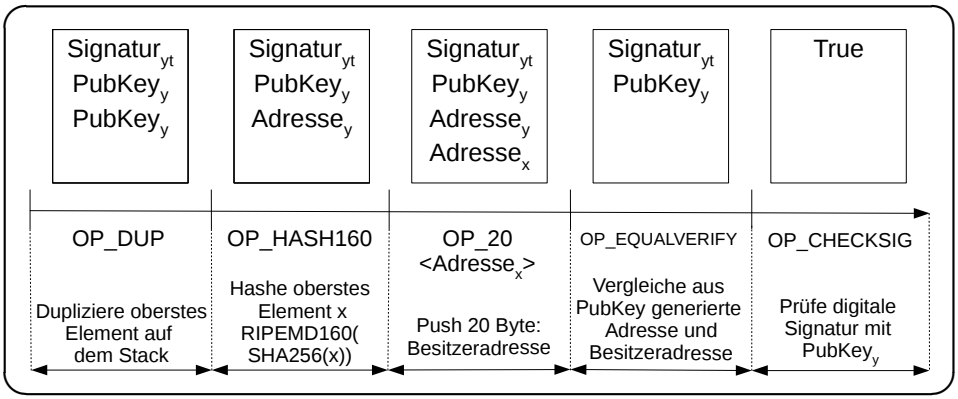


Abb. 2.2.: P2PKH Aufgabenskript bei Erfolg

OP_N OP_CHECKMULTISIG
Lösung: OP_0 <Signatur$_{x1t}$> <...> <Signatur$_{xMt}$>
Anmerkung: OP_0 zu Beginn muss auf Grund eines Fehler angegeben werden.

Multisignature Skripts fordern M-aus-N gültige digitale Signaturen. Ein 2-aus-3 Multisignature Skript gibt somit potentiell drei berechtigte Besitzer an, wobei von zwei die digitale Signatur gefordert wird um das UTXO auszugeben.
Das Lösungsskript legt eine Null (wegen eines Fehlers im Protkoll) und M Signaturen auf den Stack. Die Reihenfolge der Signaturen muss der Reihenfolge der öffentlichen Schlüssel in dem Aufgabenskript entsprechen. Das Aufgabenskript prüft anschließend mittels jeden öffentlichen Schlüssel in diesem, ob die Signaturen gültig sind. Wenn eine Signatur nicht gültig ist, bricht das Skript die Ausführung ab. Wenn es zu wenig davon gibt, bricht das Skript ebenfalls die Ausführung ab. In beiden Fällen wird somit die Ausgabe des UTXO verweigert. Nur wenn die richtige Anzahl an gültigen Signaturen gegeben ist, wird die Ausgabe des UTXO gestattet.

### 2.2.3. 3. Beispiel: Belohnung für das Knacken von Hashes

Ziel: Belohnung für das finden der Rohdaten dreier Hashwerte.
Aufgabe: OP_RIPEMD160 OP_20 <Hash$_1$> OP_EQUALVERIFY

OP_RIPEMD160 OP_20 <$Hash_2$> OP_EQUALVERIFY
OP_RIPEMD160 OP_20 <$Hash_3$> OP_EQUAL
Lösung: OP_PUSHDATA* <Größe> <$Rohdaten_3$>
OP_PUSHDATA* <Größe> <$Rohdaten_2$>
OP_PUSHDATA* <Größe> <$Rohdaten_1$>
Anmerkung: Dieses Skript folgt nicht den Standards [Comb] und wird wahrscheinlich von den meisten Netzwerkknoten abgelehnt. Es ist nichtsdestotrotz ein gültiges und funktionsfähiges Skript. OP_PUSHDATA* kann OP_PUSHDATA1, OP_PUSHDATA2 oder OP_PUSHDATA4 sein. Diese Operatoren sind dazu da, Daten auf den Stack abzulegen. Der Suffix (1, 2 oder 4) gibt an, wie viele Bytes benötigt werden um die Größe der Daten anzugeben, die anschließend auf den Stack abgelegt werden.

Die Funktionsweise dieses Skripts ist ziemlich simpel. Das Lösungsskript legt zunächst die Rohdaten auf dem Stack ab, dabei ist die Reihenfolge wichtig. Das Aufgabenskript nimmt sich nacheinander jeweils die Rohdaten, hasht diese und vergleicht sie mit den vorgegebenen Hashes. Wenn alle drei Hashes der Rohdaten mit den vorgegebenen Hashes übereinstimmen, kann der UTXO ausgegeben werden.

### 2.2.4. 4. Beispiel: Daten speichern

Ziel: Daten persistent und unveränderbar speichern.
Aufgabe: OP_RETURN OP_<Anzahl Bytes> <Daten>
Lösung: Keine. Das erzeugte UTXO kann nicht ausgegeben werden, da die Aufgabe unlösbar ist.
Anmerkung: <Anzahl Bytes> ist eine Zahl zwischen 1-75, für 76-80 Bytes: OP_PUSHDATA1 76-80 anstatt OP_<Anzahl Bytes>

Das Aufgabenskript dient nur der Speicherung von Daten in der Blockchain. Bei der Erstellung einer Transaktion wird ein UTXO mit 0 Bitcoin Guthaben und diesem Aufgabenskript erstellt. Jeder Netzwerkknoten der die Transaktion mit diesem UTXO empfängt, prüft im Normalfall lediglich ob es dem Standard entspricht und löscht es anschließend wieder. Das anheften dieser Transaktion an der Blockchain bewirkt implizit das die Daten gespeichert werden, da in der Transaktion das Skript mit den Daten enthalten ist.

# 3. Solidity - Turing-Vollständig

Solidity [Comc] ist eine Turing-Vollständige Sprache, entwickelt dafür auf einem höheren Abstraktionslevel als Assembler Smart Contracts in Ethereum programmieren zu können. Im Laufe dieses Dokuments wird auf den Aufbau dieser Sprache eingegangen. Weiterhin werden die Gefahren bezüglich der Entwicklung mit einer solch mächtigen Sprache erläutert. Anschließend folgt die Erklärung über die Registrierung eines Smart Contracts und die Interaktion mit diesem, dabei wird verdeutlicht wo der Smart Contract gespeichert wird, wo dessen Ausführung stattfindet und wo Zustandsänderungen durch diesen aufgezeichnet und vor allem wie diese mit dem Netzwerk synchronisiert werden.

## 3.1. Einführung

Soldity ist eine höherlevelige Sprache, deren Design von C++, Python und JavaScript beeinflusst wurde. Die Sprache soll Implementierungen von Smart Contracts (im weiteren Verlauf der Einfachheit halber „Vertrag“ genannt) praktischer gestalten, als diese in Assembler zu formulieren. Weiterhin soll sie die Lesbarkeit erhöhen und somit Review-Prozesse erleichtern. Die Sprache ist streng typisiert, ermöglicht den Umgang mit Vererbung, Sichtbarkeiten, komplexen Berechtigungsmodelle, das definieren von komplexen Datentypen, die Behandlung von Fehlerfällen und die Erstellung von Programmbibliotheken. Ist ein Smart Contract in Solidity vollendet, wird dieser vor der Verteilung im Ethereum-Netzwerk (was den Quelltext statisch werden lässt) in Bytecode umgewandelt, welche die EVM interpretieren kann. Im folgenden werden besondere Sprachelemente aufgezeigt und kurz erklärt, die für eine Sprache nicht üblich sind, jedoch in diesem besonderen Anwendungsszenario notwendig oder von besonderem Nutzen sein können. Anschließend folgt ein Abschnitt

in dem es den Umgang mit persistenten Daten geht. Abschließend werden einige Entwicklungsumgebungen vorgeschlagen, in denen die Entwicklung von Smart Contract mit Soldity bequem von statten geht.

### 3.1.1. Besondere Sprachelemente

In diesem Abschnitt geht es nicht um alle Sprachelemente, jene die eine Turing-Vollständige Sprache benötigt werden hier nicht weiter betrachtet. Es geht hierbei um besondere Sprachelemente, welche es ermöglichen komplexe Sachverhalte einfach darzustellen sowie die Erzeugung redundanten Codes vermeiden und so letztendlich dem Entwickler die Programmierung erleichtern. Weiterhin geht es um einige Sprachelemente, welche auf Grund der Architektur von Ethereum notwendig sind.

#### Komplexe Datentypen

Neben atomaren Datentypen wie Ganzzahlen, Booleans und Festkommazahlen existieren komplexere Datentypen. Dazu zählen Beispielweise Strukturen, identisch zu denen in C, Mappings (Abbildungen), Strings und Adressen. Letzteres ist ein spezieller Datentyp, der bislang nicht existierte, da er nicht benötigt wurde. Dieser bildet eine Ethereum Adresse (20 Byte) auf Eigenschaften und Funktionen bezüglich der Adresse ab. So können Informationen wie beispielsweise das Guthaben abgerufen werden oder Funktionen aufgerufen werden, die mit dem der Adresse zu Grunde liegenden Konto interagieren.

#### Zugriffsberechtigung

Solidity ermöglicht es Zugriffsberechtigungen simpel zu handhaben. Dafür stehen drei Sprachelemente zur Verfügung. Eines davon ermöglicht die Angabe von Sichtbarkeiten, ähnlich wie in Java. Ein weiteres Sprachelement ermöglicht es Funktionen einzuschränken. Diese können dann zum Beispiel den Zustand des Smart Contracts nicht ändern, keine Daten lesen oder kein Guthaben annehmen. Ein weiteres Sprachelement ermöglicht die Definition einer Funktion die Bedingungen prüft, welche Modifier genannt werden. Die Angabe des Modifiers bei einer Funktion hat zur Folge, dass diese nur ausgeführt wird, wenn die im Modifier angegeben Voraussetzungen erfüllt sind.

### Ereignisse (Protokollierung)

Ereignisse (events) ermöglichen das Definieren der Struktur von Logeinträgen und die Protokollierung von Datensätzen, die der Struktur entsprechen. Das Ereignis wird wie eine Funktion aufgerufen, wobei die Funktionsargumente die Dummyeinträge der Struktur ausfüllen müssen. Diese Ereignisse sind besonders für sogenannte „light clients“ von Vorteil, welche nicht die vollständige Blockchain von Ethereum lokal halten, da sie so eine effiziente Möglichkeit haben die Geschehnisse innerhalb eines Smart Contracts nachzuvollziehen.

### Vererbung

Verträge können von anderen Verträgen globale Variablen, Funktionen, Modifier und Ereignisse erben. Das Vererben funktioniert wie in objektorientierten Sprachen. Des Weiteren können abstrakte Smart Contracts erzeugt werden, die als Vorlage dienen, das heißt Funktionsprototypen enthalten deren Implementierung unausweichlich ist.

### Löschanweisungen

Ein Smart Contract terminiert nie. Genau wie in einem schlecht programmiertes Programm das allokierten Speicher nicht freigibt, können sich auch in Smart Contracts Anhäufungen von Datenmüll sammeln die den Speicher der Netzwerkknoten unnötig füllen. Daten die nicht mehr gebraucht werden und dennoch Ressourcen verbrauchen können mit Hilfe des Befehls „delete“ so wieder freigegeben werden.
Weiterhin gibt es eine Möglichkeit einen Smart Contract permanent zu deaktivieren, sodass Ethereum Netzwerkknoten den Smart Contract und die dazugehörigen Daten verwerfen können. Dabei wird eine Funktion aufgerufen die eine Adresse als Argument enthält, an die das Guthaben des Smart Contracts weitergeleitet wird bevor dieser deaktiviert wird. Nichtsdestotrotz ist ein Vertrag, der einmal über eine Transaktion in die Blockchain aufgenommen wurde, für immer und ewig in dieser enthalten. Wenn jemand es möchte kann er die Blockchain bis zum Zeitpunkt, an dem dieser Befehl ausgeführt wurde, verwenden um den Vertrag und dessen Zustände zu rekonstruieren. Mit diesem könnte dann bei Bedarf weiter interagiert werden, was jedoch zu einer Abspaltung der Blockchain führen würde, was letztendlich bedeutet das

die lokale Blockchain sich ab dem Zeitpunkt anders fortführt als die global anerkannte Version dieser.

### Besondere Variablen

Solidity bietet besondere Variablen an, die zu jedem Zeitpunkt zur Verfügung stehen. Diese sind zu einem dazu da, um Zeiten und (Ethereum-) Einheiten umzurechnen. Weiterhin gibt es einige davon, um Informationen zu gewinnen. Das können Informationen über einen Block, den Sender, die Uhrzeit oder der Transaktion in welcher der Smart Contract aufgerufen wurde enthalten.

### Fehlerbehandlung

Sollten im Smart Contract interne Fehler auftreten, Beispielsweise Werte die außerhalb des vorgegebenen Bereiches liegen, so können diese mit Hilfe von „assert“-Anweisungen abgefangen werden. Handelt es sich um externe Fehler, Beispielsweise eine falsche Benutzereingabe oder ein unerwarteter Rückgabewert nach der Interaktion mit einem weiteren Smart Contract, kann ein „require()“- oder „revert() “-Befehl genutzt werden, um die Ausführung abzubrechen. Da die Ausführung des Vertrages somit fehlerhaft verlief, folgt ein Zurücksetzen aller Veränderungen.

### Interaktionen mit anderen Smart Contracts

Smart Contracts können mit weiteren Smart Contracts interagieren. Das ermöglicht letztendlich Programmbibliotheken zu erzeugen. So kann Beispielsweise einmalig ein Smart Contract erzeugt und an die Blockchain gesendet werden, der eine Bibliothek zum Ausführen komplexerer Berechnungen enthält. Sobald dieser in der Blockchain festgehalten wird, erhält eine einmalige Adresse über die er erreichbar ist. Andere Smart Contracts können sich diese Adresse als Konstante definieren und anschließend Funktionsaufrufe an diesen durchführen, um so die komplexen Berechnungen auszulagern und den Code übersichtlich zu gestalten.
Die Interaktion zwischen Smart Contracts geht sogar so weit, dass diese weitere Smart Contracts erzeugen können.

#### Fallback Funktion

Wenn ein nicht vorhandene Funktion eines Smart Contracts aufgerufen wird oder gar keine Daten an den Vertrag übermittelt werden, wie verhält sich der Vertrag dann? Fallback Funktionen ermöglichen für solche Fälle einen Standardablauf anzugeben. Diese Funktionen werden in solchen Fällen aufgerufen und können dann z.B. das Ereignisse protokollieren oder dem Sender einige Gebühren ersparen.

### 3.1.2. Entwicklungsumgebungen

Im Grunde genommen kann Solidity in jedem Textbearbeitungsprogramm verfasst werden. Anwendungen, die unter Anderem dazu entwickelt wurden mit Smart Contracts aus dem Ethereumnetzwerk zu interagieren, bieten üblicherweise die Möglichkeit an eigene Smart Contracts zu kompilieren und serialisieren. Des Weiteren ist es in solchen Anwendungen zumeist möglich eigene Smart Contracts im Ethereumnetzwerk zu registrieren (mehr dazu in Kapitel 3.4). Trotzdem kann es Sinnvoll sein eine Entwicklungsumgebung zu nutzen, da diese Zusatzfunktionen wie Beispielsweise die statische Codeanalyse oder die Kompilierung und interne Ausführung anbieten. Über den Webbrowser kann unter https://remix.ethereum.org/ eine Entwicklungsumgebung für den Browser verwendet werden, welche die zuvor genannten Zusatzfunktionen anbieten. Wer lokal Entwickeln möchte kann z.B. Atom verwenden. Atom ist ein mittels Plugins modifizierbares Textbearbeitungsprogramm. Die notwendigen Plugins für Solidity sind dafür bereits vorhanden.

## 3.2. Ethereum Virtual Machine

In Solidity geschriebener Code wird bei der Kompilierung in Binärdaten umgewandelt. Der Prozess des veröffentlichen eines Smart Contracts beinhaltet, dass dieser persistent und unveränderbar gespeichert wird, was zur Folge hat das der aus Solidity resultierende Binärcode nach der Veröffentlichung unveränderbar ist. Die EVM ist dazu da, dass jeder Netzwerkknoten unabhängig von der Prozessorarchitektur in der Lage ist den Binärcode auszuführen und entsprechende Zustandsänderungen vorzunehmen. Dadurch das jeder Netzwerkknoten mit Hilfe der EVM in der Lage ist Turing-Vollständige Smart Contracts

auszuführen und die Ergebnisse dieser zwischen den Netzwerkknoten abgeglichen werden, kann das Ethereumnetzwerk als dezentraler Computer gesehen werden. Im Folgenden werden die Eigenschaften der EVM [Woo14] aufgezählt und beschrieben. Besonders Interessant ist die Eigenschaft, dass die EVM genau genommen nur quasi-Turingvollständig ist, da die erfolgreiche Ausführung des Codes von einem externen Faktor abhängt, dem Gas.

### 3.2.1. Eigenschaften

Die EVM ist Stackbasiert, wobei der Stack eine Tiefe von 1024 Wörtern hat. Die Wortgröße beträgt 256 bit (32 Byte). Das liegt zu einem daran, dass Ethereum den Keccak256 Hashalgorithmus verwendet, welcher eine zentrale in Ethereum hat und ein 256 bit langes Ergebnis erzeugt. Weiterhin erzeugen verwenden die elliptischen Kurven in Ethereum 256 bit lange Zahlen. Die Zahlen werden im Big-Endian format repräsentiert. Wie bereits zu Beginn dieses Kapitel erwähnt wurde, ist die EVM quasi-Turingvollständig, da die erfolgreiche Ausführung von einem externen Faktor abhängt. Dies wird genauer im folgenden Kapitel erklärt. Der Speicher ist ein einfaches Wort-Adressierbares Byte-Array. Die EVM verwendet keine klassische Von-Neumann-Architektur. Der Programmcode (der Smartcontract) liegt in einem virtuellen Read-Only Memory (ROM), der nur durch eine spezielle Instruktion ausgelesen werden kann. Ausführungsfehler führen in der EVM dazu, dass alle Zustandsänderungen die aus dem Smart Contracts hervorgehen unwirksam werden.

### 3.2.2. Gas als Antriebsmittel

Das Gas wird als Antriebsmittel für die EVM benötigt. Jede Instruktion ist ein Gaswert zugewiesen, der abbildet wie aufwendig die Instruktion ist. Stellt nun jemand eine Transaktion zusammen die einen Aufruf an einen Smart Contract enthält, so muss die Person genügend Gas für die Ausführung zur Verfügung stellen. Wenn das Gas nicht ausreicht bricht die EVM die Ausführung ab, die Zustandsänderungen werden unwirksam und das gesamte Gas wird konsumiert. Gas wird mit Ether bezahlt, die Währung in Ethereum. Üblicherweise simuliert man die Ausführung des Smart Contracts, zählt die Instruktionen und berechnet abschließend die Gaskosten für die Ausführung derer. Anschließend berechnet man, wie viel Ether man für das Gas zur Verfügung stellt:

$$c_{Ether} = Gasmenge \cdot Gaspreis \tag{3.1}$$

Der Gaspreis variiert und hängt von der Netzwerklast des Ethereumnetzwerks ab. Zum Zeitpunkt an dem dieses Dokument verfasst wurde kostet Gas 4 GWei, das sind $4 \cdot 10^{-9}$ Ether, wenn eine Transaktion im Schnitt innerhalb einer halben Minute durchgeführt werden soll. Ein Ether kostet ca. 650 Euro, der Preis variiert ebenfalls (stark).

#### Beispiel - Gaskosten - Transaktion ohne Daten

Die Instruktion eine Transaktion zu verarbeiten kostet 21.000 Gas. Das ist der Mindestbetrag an Gas, den jemand zur Verfügung stellen muss um eine Transaktion an das Netzwerk zu propagieren. Somit ergibt sich bei einem Gaspreis von 4 GWei folgende Gebühr:

$$\begin{aligned} c_{Ether} &= (21 \cdot 10^3) \cdot (4 \cdot 10^{-9} Ether) \\ &= 84 \cdot 10^{-6} Ether \end{aligned} \tag{3.2}$$

Umgerechnet in Euro kostet eine Transaktion bei einem Gaspreis von 4 GWei somit mindestens:

$$\begin{aligned} c_{Euro} &= 84 \cdot 10^{-6} Ether \cdot 650 \frac{€}{Ether} \\ &= 0,0546€ \end{aligned} \tag{3.3}$$

#### Beispiel - Gaskosten einer Addition

Eine Addition kostet 3 Gas. Bei eine Gaspreis von 4 Gwei ergibt das folgende Kosten:

$$\begin{aligned} c_{Ether} &= 3 \cdot (4 \cdot 10^{-9} Ether) \\ &= 12 \cdot 10^{-9} Ether \end{aligned} \tag{3.4}$$

Umgerecht in Euro bei einem Preis von 650 Euro pro Ether:

$$\begin{aligned} c_{Euro} &= 84 \cdot 10^{-6} Ether \cdot 650 \frac{€}{Ether} \\ &= 0,0000078€ \end{aligned} \tag{3.5}$$

## 3.3. Gefahren

Die Mächtigkeit von Smart Contracts führt auch zu Gefahren. Programmierfehler im Smart Contract oder unentdeckte Fehler der EVM könnten massive Auswirkungen haben. Gerade Smart Contracts die riesige Summen Ethereum von Vertragsteilnehmern anhäufen, können so zu einer tickenden Zeitbombe werden. In den nächsten Abschnitten geht es um direkte Angriffsszenarien, die durch die Turing-Vollständigkeit von Ethereum in Frage kommen. Weiterhin werden Angriffsszenarien auf Grund von Programmierfehler und Fehlern.

### 3.3.1. DDos Angriffe

Im folgenden werden zwei Szenarien präsentiert die einen DDos Angriff darstellen. Ein Szenario ergibt sich direkt aus der Turing-Vollständigkeit der EVM. Ein weiteres Szenario war ein Resultat aus einem sehr begehrten Smart Contract.

#### Beabsichtigt

Wenn eine Transaktion die einen Aufruf an einen Smart Contract enthält an das Netzwerk propagiert wird, validiert jeder vollständige Netzwerkknoten diesen Aufruf. Das bedeutet, dass jeder Knoten den Funktionsaufruf durchführt. Angenommen der Funktionsaufruf enthält eine Endlosschleife oder eine Schleife, in der Unmengen an Daten in den Hauptspeicher oder den Festplattenspeicher geschrieben werden. Würde das jeden Netzwerkknoten außer Gefecht setzen? Die Antwort ist nein. Da die Ausführung jeder Instruktion Gas kostet, ist irgendwann einfach das Gas aufgebraucht, der Funktionsaufruf wird als fehlgeschlagen gekennzeichnet und der Zustand wird vor den Funktionsaufruf zurückgesetzt. Der Angreifer hat somit in diesem Szenario lediglich sein Guthaben verbrannt.

#### Unbeabsichtigt

Es existiert ein Smart Contract mit dem Namen „CryptoKitties“. Dieser Smart Contract bildet ein Kartenspiel ab, indem es um Katzen geht die sich fortpflanzen können und so eine neue Katze mit einzigartigen Eigenschaften (Farbe, Muster, etc.) erzeugen. Der Smart-Contract erledigt dabei die Aufbewahrung,

das Fortpflanzungsverhalten und den Handel. Die Katzen werden durch Gene (eine Zahl) definiert, die eine externe Applikation zu Grafiken umwandeln kann. Der Vertrag ist unter der Adresse 0x06012C8CF97BEAD5DEAE237070F9587F8E7A266D gespeichert. Mit einem Blockexplorer (Anwendung zum durchsuchen der Blockchain) kann man den Quellcode begutachten.
Abgesehen davon, dass der Smart Contract auf eine einzigartige Weise die Möglichkeiten der Smart Contracts aufzeigt, hatte er einen weiteren Effekt auf Ethereum. Im Dezember 2017 wurde dieses Spiel so beliebt, dass die Interaktionen mit dem Vertrag gewaltige Ausmaße annahmen. Die Anzahl der Transaktionen war so groß, dass das Netzwerk überlastet wurde. Viele Transaktionen verweilten über mehrere Stunden oder gar Tagen im Netzwerk und wurden nicht durchgeführt. Das lag daran, dass die Zahl der ausstehenden Transaktionen zu hoch war und die Benutzer von Ethereum begannen sich mit den Transaktionsgebühren zu überbieten. Die Transaktionsgebühren stiegen so in kürzester Zeit um das fünffache. Ältere Transaktionen die im Vergleich dazu geringe Transaktionsgebühren beinhalteten kamen so nicht an die Reihe. Erst als der Rummel nachließ konnte der Überhang älterer Transaktionen verarbeitet werden. Es ist zwar Möglich eine Transaktion die noch nicht in die Blockchain hinzugefügt wurde rückgängig zu machen, allerdings ist der Prozess kein Standardvorgehen und den wenigsten bekannt. Weiterhin fallen trotzdem Transaktionsgebühren an.

### 3.3.2. Vollständige Pfadabdeckung

Der Code von Smart Contracts kann sehr komplex werden. Insgesamt gibt es unheimlich viele Ausführungspfade die ein Smart Contract durchführen kann. Da der Code statisch ist und zudem Guthaben von Vertragsteilnehmer beinhalten könnte, sollte dieser extrem Sorgfältig verfasst und geprüft werden. Immer wieder kam es in der Vergangenheit dazu, dass Guthaben in Smart Contracts gestohlen oder eingefroren wurde. Einige dieser Vorfälle werden im Folgenden beschrieben.

#### 1. 2016 - DAO Hack

Im Jahr 2016 erzeugte Visionäre in Ethereum einen Smart Contract, den Decentralized Autonomous Organization (DAO) Vertrag, der dazu da war Projekte in Ethereum zu fördern. Investoren konnten Guthaben an den Smart

Contract überweisen und erhielten anschließend ein Stimmrecht, dass dazu dient sich für oder gegen zukünftige Projekte auszusprechen. Viele Investoren waren fasziniert von dieser Idee und partizipierten, sodass die von den Entwicklern geschätzte Investitionssumme von 500 Tausend Dollar weit überstiegen wurde. Insgesamt betrug diese ca. 68 Millionen Dollar.
Ein geschickter Hacker erkannte nicht offensichtliche Sicherheitslücken im Code und konstruierte einen mehrstufigen Angriff, mit dem es ihm gelungen ist das gesamte Guthaben des Vertrags für sich zu beanspruchen. Dieser Hack hatte maßgebliche Auswirkungen auf Ethereum.

### 2. 2017 - Crowdfunding & Parity Selbstzerstörung

Auf Grund von schlecht programmierten Smart Contracts, die beispielsweise notwendige Voraussetzungen im Code nicht prüfen oder Variablen nicht initialisieren, kam es im Jahr 2017 bei mindestens zwei Smart Contracts zu einer ungenehmigten Zerstörung dieser. Die Zerstörungsfunktion eines Smart Contract hat zur Folge, dass das gesamte Guthaben darin an eine Adresse überwiesen wird und der Vertrag als ungültig markiert wird. Diese Funktion wurde bei zwei großen Smart Contracts auf Grund eines Programmierfehlers unberechtigt aufgerufen, was zur Folge hatte das in einem Crowdfunding Vertrag ca. 150 Millionen Dollar und in einem Smart Contract zur Verwaltung von Guthaben ca. 280 Millionen Dollar unberechtigt aus den Verträgen entnommen und diese anschließend zerstört wurden.

### Analyse und Maßnahmen

Ein Paper [Nik+18] wurde im Februar 2018 veröffentlicht, welches die Ergebnisse eine Analysetools dokumentiert, dass Smart Contracts auf Sicherheitslücken untersucht. Dieses Tool wurde dazu entwickelt im großen Maße automatisch Smart Contracts in der Blockchain zu finden und diese auf Sicherheitslücken zu untersuchen. Dabei sucht das Tool nach Möglichkeiten Guthaben einzufrieren, Guthaben unberechtigt zu entnehmen und zuletzt den Vertrag zu zerstören. Das Analysetool stellte fest, das von ca. eine Millionen Verträgen insgesamt 34200 Sicherheitslücken enthalten. Dabei gilt zu beachten, dass ungefähr 70% davon true-positives sind.
Die Schlussfolgerung aus den erfolgreichen Angriffsszenarien der Vergangenheit und der Analyse mit Hilfe des MAIAN Tools ist eindeutig. Wer Smart

Contracts entwickelt und diese veröffentlicht, sollte sehr achtsam und bedacht bei der Entwicklung dieser Vorgehen. Wenn diese nicht nur in kleinen Kreisen verwendet werden und zudem größeres Guthaben enthalten, sind Analysen, eine formale Verifikation und Reviews von Fachkollegen vor der Veröffentlichung des Vertrages ein angemessenes und vernünftiges Verfahren.

### 3.3.3. Weitere Fehler

Monero Gold ist ein Smart Contract für Ethereum, welcher eine eigene Währung (Tokens) in diesem enthält. Die Funktion des Smart Contracts ist es diese Tokens zu verwalten und den Austausch zwischen Benutzern zu ermöglichen. Der Entwickler ließ die interne Prüfung einiger Werte außen vor, sodass es möglich war eine 256 bit lange vorzeichenlose Ganzzahl im Smart Contract, die das Guthaben bezüglich einer Adresse aufbewahrt, zu einem Überlauf zu bringen. Dieser Überlauf sorgte dafür, dass der Entwickler (der zugleich der Angreifer war) anschließend $2^{256} - 1$ dieser Tokens besaß, die er dann in einer Tauschbörse verkaufte.
Die Frage die dieses Szenario aufwirft ist, wo der Fehler entstanden ist. Einerseits war es ein Fehler im Smart Contract, da dieser Ausnahmefall nicht überprüft wurde. Andererseits kann man sich jedoch auch die Frage stellen, ob so ein Zahlenüberlauf nicht in der EVM als Fehlerfall betrachtet werden sollte und diese auf Grund dessen die Ausführung abbrechen und die Änderungen zurücksetzen sollte.

## 3.4. Registrierung eines Smart Contract

Zunächst wird der in Solidity geschrieben Smart Contract in einen für die EVM interpretierbaren Binärcode kompiliert. Anschließend folgt eine Serialisierung dessen, sodass dieser einem im Protokoll vorgesehen Format entspricht. Nun ist der Smart Contract bereit an das Ethereumnetzwerk verbreitet zu werden. Die Registrierung eines Smart Contracts erfolgt mit einer speziellen Transaktion, welche den kompilierten und serialisierten Code enthält. Die Transaktion enthält weiterhin genügend Gas, um den Vertrag zu registrieren und zu initialisieren. Diese kann anschließend an das Ethereumnetzwerk propagiert werden. Bei der Verteilung simuliert jeder Knoten ob genügen Gas zur Verfügung steht um diesen Code zu initialisieren und ob diese problemlos

verläuft. Nach der Simulation setzt dieser alle Zustandsänderungen zurück. Wenn die Initialisierung fehlerhaft verläuft oder das Gas nicht reicht, lehnt der erste Netzwerkknoten der die Transaktion empfängt diese ab. Sollte alles stimmen wird die Transaktion über das Ethereumnetzwerk verteilt und wartet anschließend auf Integration in einen Block. Das anfügen dieses Blocks an die Blockchain bedeutet Schlussendlich, das der Smart Contract akzeptiert und offiziell publiziert wurde. Jeder Netzwerkknoten der den Block empfängt aktualisiert seine State DB entsprechend:

1. Account für den Smart Contract hinzufügen (enthält Referenz auf Code und Daten)
2. Code des Smart Contract hinzufügen
3. Persistenten Speicher anlegen

## 3.5. Interaktion mit einem Smart Contract

Nachdem der Smart Contract erfolgreich im Ethereumnetzwerk verewigt wurde, kann mit diesem interagiert werden. Dabei unterscheidet man zwischen lesenden Funktionen, die lediglich Daten auslesen und zurückgeben und schreibende Funktionen, die den Zustand des Smart Contracts ändern. Die zwei nächsten Abschnitte beschäftigen sich mit diesen Funktionen.

### 3.5.1. Lesezugriff

Alle Funktionen eines Smart Contracts sind in der State DB abgespeichert. Ein vollwertiger Netzwerkknoten kann diese Funktionen allesamt intern ausführen und die Resultate betrachten. Wenn die Funktionen während der Durchführung den Zustand der Vertrages nicht ändern, gibt es keine Notwendigkeit diese Interaktion an das Netzwerk mitzuteilen. Der Zugriff bleibt somit lokal und ist mit keinen weiteren Kosten verbunden.

### 3.5.2. Schreibzugriff

Ein Schreibzugriff auf einen Smart Contract ändert den Zustand dessen. Ist es erwünscht das die Zustandsänderung in allen Netzwerkknoten anerkannt und durchgeführt wird, so ist es Notwendig einen Funktionsaufruf an den Smart

Contract in einer Transaktion festzuhalten. Der Smart Contract sowie die aufgerufene Funktion werden dabei über eindeutige Hashwerte referenziert. Für die Angabe der Daten (z.B. der Funktionsargumente) ist in der Transaktion ein Feld vorgesehen. Bevor die Transaktion zusammengestellt wird, muss bekannt sein wie viel Gas die Transaktion und die darin enthaltene gewünschte Ausführung des Smart Contracts konsumiert. Dafür wird die Ausführung ein Mal simuliert und dabei gezählt, wie viel Gas die Ausführung kostet. Anschließend werden alle Zustandsänderungen, die durch die Simulation bewirkt wurden, wieder rückgängig gemacht. Die Transaktion kann daraufhin mit einem ausreichenden Betrag an Gas erstellt und an das Netzwerk propagiert werden. Sobald die Transaktion in der Blockchain aufgenommen wurde, führt jeder Netzwerkknoten den darin angegeben Funktionsaufruf aus und übernimmt folglich die gewünschten Zustandsänderungen.

## 3.6. Beispiele

Für diese Dokumentation wurden zwei Beispielverträge entwickelt. Der erste Vertrag soll ein einfaches Beispiel darstellen, in dem keine komplexeren Abläufe enthalten sind. Dieses dient als Einstieg.
Der zweite Vertrag stellt ein komplexeres Beispiel dar, in dem es um die Kaffeeabrechnung, das aufnehmen von Kundenwünschen und die automatische Bestellung von neuem Kaffee geht. Dieses Beispiel ist wesentlich komplexer und erfordert zudem die Interaktion mehrerer Smart Contracts untereinander.

### 3.6.1. Hello World

Der Hello World Smart Contract hat die Funktion bei dessen Initialisierung einen Besitzer festzulegen. Dieser hat anschließend die Berechtigung einen neuen Besitzer zu deklarieren oder den Smart Contract zu zerstören. Die Funktion des Smart Contracts ist es, zu zählen wie oft eine Adresse eine Funktion des Smart Contracts aufgerufen hat und diesen Wert an den aufrufenden zurück geben zu können. Der Vertrag ist im Anhang A.1 hinterlegt und besteht aus folgenden Komponenten:
Globale Variablen:

- address private owner - Die Adresse des Besitzers des Smart Contracts.

- mapping(address => uint) public hicount - Abbildung zwischen Adresse und vorzeichenlose Ganzzahl. Wird zum zählen der Hello World - Aufrufe verwendet.

Modifier:

- modifier byOwner() - schränkt damit gekennzeichnete Funktionen so ein, dass sie nur ausführbar sind wenn sie vom Besitzer ausgeführt werden.

Funktionen:

- function HelloWorld() public - Konstruktor, deklariert den Besitzer.
- function delegateOwner(address newowner) public byOwner() - Besitzer ändern, kann nur vom aktuellen Besitzer aufgerufen werden.
- function killMe() public byOwner() - Zerstört den Smart Contract, überweist das darin enthaltene Guthaben an den Besitzer.
- function getHi() public view returns (uint) - Read-Only Funktion, gibt aus wie oft der Aufrufende sayHi() ausgeführt hat.
- function sayHi() public returns (uint) - Hello World Zähler des Aufrufenden.

### 3.6.2. Kaffeevertrag Smart Contract & Kaffeegeschäft Smart Contract Interface

Dieses Projekt besteht aus zwei Smart Contracts. Ein Smart Contract dient dazu Benutzern das Einkaufen von Kaffee über diesen abzuwickeln und dabei eine Stimme für den Kaffee ihrer Wahl abzugeben. Weiterhin bestellt der Smart Contract ab einem bestimmten Grenzwert an verkauften Kaffees neuen Kaffee automatisch, wofür ein Verkäufer einen Smart Contract implementieren muss der das Kaffeegeschäft Interface implementiert. Dieser Smart Contract wird im weiteren Verlauf „Kaffeevertrag“genannt und ist als Anhang A.2.1 diesem Dokument beigefügt worden. Der zweite Vertrag enthält nur Datenstrukturen und Schnittstellen, die ein Kaffeegeschäft in ihren Smart Contract implementieren müssen, damit ersterer Smart Contract von diesem Kaffee bestellen kann. Dieser wird im folgenden „Kaffeegeschäft“ genannt und ist im Anhang A.2.2 enthalten. Es folgt zunächst die Beschreibung des Kaffeegeschäfts

### Schnittstelle: Kaffeegeschäft

Das Kaffeegeschäft muss mindestens drei Datenstrukturen unterstützten. Diese sind im Quellcode unter „contract CoffeeOrder“ und „contract CoffeeShopData“ definiert. Zwei davon, „struct struct_simpleorder“ und „struct struct_complexorder“ werden benötigt um Bestellungen in einer einheitlichen Form an das Kaffeegeschäft übertragen zu können. „struct struct_coffee“ wird benötigt, um von Geschäften einheitlich Daten über eine Kaffeesorte und so letztendlich auch über das gesamte Sortiment abzurufen. Die Vorgegebenen Funktionen sind folgende:

- function coffeeStoreAlive() [...] - dient dazu zu prüfen, ob dieser Smart Contract ein Kaffeegeschäft ist.
- function getPrice(uint32 id) [...] - Der Preis des Kaffees mit der Identifikationsnummer id wird zurückgegeben.
- function simpleCoffeeOrder(...) [...] - Dient dazu Kaffee zu bestellen, entweder einen Kaffee an eine Adresse oder mehrere Kaffees an optional mehrere Adressen.
- function getCoffee(uint32 id) [...] - Gibt Daten (struct_coffee) zum Kaffee mit der Identifikationsnummer id zurück.
- function getCoffeeDB() [...] - Gibt die gesamte Kaffeedatenbank zurück (struct_coffee[]).

Wie die Datenbank aller Kaffees intern gehalten wird ist dem Entwickler überlassen, der diese Schnittstelle implementiert. Weiterhin steht es frei, wie die Verwaltung der Kaffeesorten von statten geht.

### Kaffeevertrag

Angenommen wir besitzen eine Kaffeemaschine, zum Beispiel in der Küche der Hochschule, welche wir mit diesem Smart Contract betreiben möchten. Die Idee ist, dass ein Benutzer der Kaffee von dieser Kaffeemaschine möchte an einen Smart Contract einen festgelegten Preis für Kaffeeüberweisen kann, um sich einen Kaffee von dem Automaten ausgeben zu lassen. Der Smart Contract enthält die Bezahlung und Protokolliert dieses Ereignis, wobei das Protokoll von der Kaffeemaschine eingesehen wird. Der Automat gibt anschließend nach

erfolgter Zahlung den Kaffee aus. Das ist allerdings noch nicht die gesamte Funktionalität des Vertrages. Dieser Vertrag bestellt automatisch neuen Kaffee, wenn eine gewisse Anzahl an Kaffees vom Automaten gekauft wurden. Der Smart Contract kann dabei aus einer modifizierbaren Liste die Adressen des Smart Contracts entnehmen, welche die Kaffeegeschäft-Schnittstelle implementiert haben. Die Auswahl des zu bestellenden Kaffees wird dabei durch die Benutzer des Automaten bestimmt: Immer wenn ein Benutzer einen Kaffee bestellt, ruft der Automat eine Funktion des Kaffeevertrages auf und lässt sich die Kaffeedatenbanken aller anerkannten Kaffeegeschäften ausgeben. Dieser bietet daraufhin den Benutzer die Möglichkeit, seine Stimme für einen der Kaffeesorten abzugeben, welche zusätzlich durch die Anzahl der Kaffees die dieser Bestellt hat gewichtet wird. Somit hat ein Benutzer Einfluss darauf, welche Sorte demnächst automatisch bestellt wird. Für die Verwaltung der Geschäfte, des Einzelpreises für Kaffee und dem Grenzwert an Kaffees für eine automatische Bestellung wurde zusätzlich ein Berechtigungsmodell implementiert, das es ermöglicht Verwalter zu bestimmen.
Der Kaffeevertrag enthält weitere Smart Contracts, welche für diesen notwendig sind:

**contract LogBook**
Dieser Smart Contract stellt die Struktur aller möglichen Protokolleinträge dar. Mit diesem Protokolleinträgen ist es dem Automaten einfach möglich das Resultat der Interaktion mit diesem Smart Contract herauszufinden. Weiterhin sind Vorlangen für Änderungen bezüglich der Verfügbaren Geschäfte, des Preises, usw. vorhanden, sodass stets transparent ist, wer welche Änderungen am Smart Contract vorgenommen hat.

**contract CoffeeOwner**
Hier wird der Besitzer und dessen Möglichkeiten definiert. Der Besitzer kann den Vertrag zerstören, das Guthaben abheben, einen anderen Besitzer und er kann Verwalter bestimmen. Der Besitzer ist automatisch selber ein Verwalter. Die Möglichkeiten der Verwalten sind im folgenden Smart Contract angegeben.

**contract CoffeeManager**
Verwalter sind dazu da den Smart Contract zu konfigurieren und Notfalls manuelle Bestellungen durchzuführen. Dieser Vertrag gibt den Verwaltern die

Berechtigung Kaffeegeschäfte hinzuzufügen und zu entfernen, den Preis eines einzelnen Kaffees festzulegen, den Grenzwert an Kaffeebestellungen vor einer automatischen Bestellung festzulegen, Die Lieferadresse für Bestellungen zu ändern und manuell Bestellungen durchzuführen.

**contract CoffeeOrder**
Dieser Vertrag enthält Strukturen die für eine einheitliche Bestellung notwendig sind. Diese sind identisch zu denen im Kaffeegeschäft Smart Contract.

**contract CoffeeData**
Hier sind Datenstrukturen und Konfigurationen enthalten. Die Konfigurationen bestehen aus einer Liste von zugelassen Kaffeegeschäften, einen Grenzwert an gekauften Kaffees bevor eine automatische Bestellung durchgeführt wird, die Anzahl der gekauften Kaffees, den Preis für einen gekauften Kaffee und die Lieferadresse für neuen Kaffee. Die Datenstrukturen geben an, wie Informationen zu einer Kaffeesorte dargestellt werden, wie Informationen über alle Kaffeesorten jedes akzeptierten Kaffeegeschäftes aussehen und wie Abstimmungen für die Bestellung des nächsten Kaffees abgespeichert werden.

**contract CoffeeContract**
Dies ist der primäre Smart Contract. Er enthält alle in diesem Abschnitt beschriebenen Smart Contracts. Somit kennt er alle Datenstrukturen und Konfigurationsparameter, alle Protokollvorlagen sowie das Berechtigungsmodell für den Besitzer und die Verwalter. Hier sind zwei wesentliche Funktionen enthalten, eine dient dazu das gesamte Kaffeesortiment aller anerkannten Kaffeegeschäfte abzufragen und zurückzugeben und eine weitere Funktion dient dazu bei der Bestellung von Kaffee am Kaffeeautomaten die Abstimmung des Benutzers aufzunehmen und, wenn der Grenzwert erreicht wurde, automatisch die Kaffeesorten mit den meisten Abstimmungen zu bestellen. Folgende Funktionen sind enthalten:

- function getCoffeeSorts() [...] - Gibt Datenbank mit dem gesamten Sortiment der anerkannten Kaffeegeschäfte zurück.
- function buyCoffee() [...] - Prüft Bezahlung von Kunden, nimmt Stimme für nächste Kaffeesorte auf, Bestellt bei erreichen des Grenzwertes automatisch neuen Kaffee basierend auf den Abstimmungen der Kunden.

- function autoOrder() [...] - Helferfunktion zum automatischen Bestellen von Kaffee.
- function deserializeCoffeeDB(...) [...] - Helferfunktion, notwendig um vom Kaffeegeschäft empfangene Daten zu deserialisieren. Notwendig, da Solidity gewisse Datenstrukturen nicht über entfernte Vertragsaufrufe zurückgeben kann.

# 4. Fazit

In diesem Dokument wurde die Entwicklung der Programmierung in Kryptowährungen dem Leser etwas näher gelegt. Dieses Kapitel schließt diese Dokumentation mit einer kurzen Zusammenfassung der Erkenntnisse ab.

## 4.1. Stand der Dinge

Die Skriptsprache in Bitcoin wird nach wie vor dafür benötigt um den Transfer von Guthaben an eine Bedingung zu binden. Obwohl die Skriptsprache sich etwas geändert hat, ist diese immer noch stark eingeschränkt. Durch Neuerung war es zwar möglich Bedingungen zu formulieren die zuvor nicht oder nur eingeschränkt möglich waren, z.B. Daten persistent zu speichern oder mehrere digitale Signaturen zu fordern, allerdings konnten viele Bedingungen in Skript nicht formuliert werden. Um den Transfer von Guthaben an komplexe Abläufe zu binden bedarf es ein mächtigeres Konzept, welches in Ethereum erstmals zumgesetzt wurde: Smart Contracts. Kurzgefasst ermöglichen diese die Bedingungen für eine Transaktion beliebig, jedoch im Rahmen der Berechenbarkeit, zu formulieren.

### 4.1.1. Weitgehend unendecktes Potential

Smart Contracts sind ein sehr junges Konzept. Verträge wie der DAO, Tokenverträge oder sogar Spiele wie Cryptokitties demonstrieren bereits die Mächtigkeit dieses Konzepts. In etwas mehr als zwei Jahren haben etliche Projekte die Möglichkeiten dargestellt, dennoch war das nur der Anfang von dem was noch kommen könnte. Möglicherweise wird Ethereum es ermöglichen, vollständig autonome Unternehmen zu realisieren. Die Mächtigkeit bringt allerdings Gefahren mit sich, welche diese unzuverlässig machen.

### 4.1.2. Unzuverlässigkeit

Die Entwicklung von Smart Contracts kann ebenso kritisch wie die Entwicklung von kryptographischen Algorithmen gesehen werden. Es gibt etliche Angriffsvektoren auf Smart Contracts, die durch die kleinste Unachtsamkeit bereits ermöglicht werden. Dieses Dokument hat viele existierende Smart Contracts präsentiert, die von einem fähigen Entwicklerteam durchgeführt worden sind und trotzdem auf Grund von Sicherheitslücken zu einem massiven Verlust an Kapital führten. Unternehmen werden diese Technologie nicht verwenden, solange sie bei Verwendung dieser stets fürchten müssen das eine Sicherheitslücke wesentliche Anteile ihres Kapitals in Gefahr bringt und somit deren Existenz gefährdet.

### 4.1.3. Skalierungsschwierigkeiten

Ethereum kann nur eine Obergrenze an Transaktionen pro Zeiteinheit durchführen. Wenn Ethereum in der Lage sein soll viele nützliche Smart Contracts praktikabel zu verwalten, müssen noch einige Ideen aufkommen und umgesetzt werden die eine Überlast des Netzwerkes verhindern. Wie am Beispiel von Cryptokitties zu sehen ist, kann bereits ein einziger sehr beliebter Smart Contract das Netzwerk auf Grund von einer zu hohen Anzahl an Transaktionen pro Sekunde überlasten.

## 4.2. Ausblick

Dieses Dokument wird mit diesem Abschnitt abgeschlossen. Einige letzte Ausblicke werden hier dargestellt, welche die Entwicklung von Ethereum und Smart Contracts betreffen und in Zukunft absehbar sind.

### 4.2.1. Sharding

Das Ziel von Sharding ist die Anzahl der Transaktionen pro Sekunde zu erhöhen und die Größe der State DB für Netzwerkknoten zu reduzieren. Dabei werden Transaktionen und Daten über verschiedene Netzwerkteilnehmer verteilt, sodass letztendlich nicht jeder Netzwerkknoten alle Daten und Transaktionen zur Verfügung hat und dadurch die Transaktionsgeschwindigkeit steigt,

allerdings die Sicherheit des Netzwerks nicht darunter leidet. Weitere Details sind im Github-Projekt von Ethereum zu finden [Comd].

### 4.2.2. Analyse & Verifikation von Smart Contracts

Die Gefahren der Smart Contracts sollten vorher analysiert werden und bestenfalls sogar durch Beweise, z.B. mittels formaler Verifikation, mit hoher Wahrscheinlichkeit ausgeschlossen werden können. Dies würde bezwecken das die Anwendung auf Grund eines geringeren Risikos für Anwender attraktiver wird. Die Entwicklung einiger solcher Tools, wie z.B. MAIAN [Nik+18] oder eines von Fujitsu entwickelten Tools [LRL18], ist bereits im Gange. Bestenfalls führt die Entwicklung solcher Tools zu sichereren Smart Contracts und führt so einem höheren Vertrauen gegenüber diesen.

# A. Anhänge

## A.1. Solidity Smart Contract - Hello World

```
1  pragma solidity 0.4.19;

3  /// @title Hello counter.

5  contract HelloWorld {

7      address private owner;

9      // State variable to count hellos
10     mapping(address => uint) public hicount;

12     // Constructor, just declare the owner
13     function HelloWorld() public {
14         owner = msg.sender;
15     }

17     // Modifier for functions only the owner should call
18     modifier byOwner() {
19         require(owner == msg.sender);
20         _;
21     }

23     // Delegate owner rights
24     function delegateOwner(address newowner) public byOwner() {
25         owner = newowner;
26     }
```

```
28      // Destroy contract
29      function killMe() public byOwner() {
30          selfdestruct(owner);
31      }

33      // Return amount of greetings
34      function getHi() public view returns (uint) {
35          return hicount[msg.sender];
36      }

38      // Greet the contract
39      function sayHi() public returns (uint) {
40          hicount[msg.sender] += 1;
41          return getHi();
42      }
43  }
```

Listing A.1: Quellcode: hello_world.sol

## A.2. Projekt: Smart Contracts für Kaffeeautomaten und Kaffeegeschäfte

### A.2.1. Solidity Smart Contract - CoffeeContract

```
1   /*
2   Smart Contract: CoffeeContract
3   Purpose: Automatically sell coffee and let the users vote what coffee they like.
4   Features: Ability to add stores, retrieve their coffee database and
5             automatically order from them depending on the votes given by the users.
6   Autor: Harald Heckmann
7   */

9   pragma solidity ^0.4.21;

11  // get some extended features including return of structs and nested arrays
12  pragma experimental ABIEncoderV2;

14  contract CoffeeData {
15    // contains the address of all accepted coffee stores
16    // since the array is public, a getter function stores(id) will be generated.
17    address[] public coffee_stores;

19    // Internal coffee configuration:
20    // How many coffees until ordering new?
21    uint32 public coffee_threshold = 100;
22    uint32 public coffee_count = 0;
23    uint128 public coffee_price = 1 finney;
24    byte[] coffee_address;

26    // Database entry from the coffee shops
```

```
27      struct struct_coffee {
28        uint32 id;
29        uint32 amount;
30        uint128 price;
31        byte[] name;
32      }

34      // Structure which contains store addresses and their coffee database
35      struct struct_coffee_overview {
36        address[] stores;
37        struct_coffee[][] coffeedb;
38      }

40      struct_coffee[] test;

42      // Mapping von Adresse auf Produktids auf votes
43      mapping (address => mapping (uint32 => uint32)) votes;
44    }

46    contract CoffeeOrder {
47      // Simple order: One specific product to one specific address
48      struct struct_simpleorder {
49        uint32 id;
50        uint32 amount;
51        byte[] destination;
52      }

54      // Complex order: Multiple products,
55      // each can be delivered to a different address
56      struct struct_complexorder {
57        uint32[] ids;
58        uint32[] amounts;
59        byte[][] destinations;
60      }
61    }

63    // Contract which contains all pre defined log entries
64    contract LogBook {
65      event UserBoughtCoffee(
66        address indexed _from,
67        uint8 _amount,
68        bool _success
69      );

71      event CoffeeAutomaticallyOrdered(
72        address indexed _shop,
73        uint32 indexed _productid,
74        uint32 _amount,
75        bool _success
76      );

78      event CoffeeOrderedByManager(
79        address indexed _shop,
80        address indexed _manager,
81        uint32 indexed _productid,
82        uint32 _amount,
83        uint128 _price,
84        bool _success
85      );

87      event ShopManagement(
88        address indexed _manager,
89        address indexed _shop,
90        bool _add,
91        bool _success
92      );

94      event ThresholdChanged(
95        address indexed _manager,
96        uint32 _old_threshold,
97        uint32 _new_threshold
98      );
```

```
100     event PriceChanged(
101       address indexed _manager,
102       uint128 _old_price,
103       uint128 _new_price
104     );

106     event AddressChanged(
107       address indexed _manager,
108       byte[] _old_address,
109       byte[] _new_address
110     );
111   }

113   // The contract containing the interface for the contract owner
114   contract CoffeeOwner is CoffeeData {
115     address internal owner;
116     mapping (address => bool) internal managers;

118     // Permission: Only executable by owner
119     modifier onlyOwner {
120       require(msg.sender == owner);
121       _;
122     }

124     // Permission: Only executable by owner and manager
125     modifier onlyManager {
126       require(msg.sender == owner || managers[msg.sender] == true);
127       _;
128     }


131     // Constructor: assign contract owner and custom address
132     function CoffeeOwner(byte[] deliveryaddress) public {
133       owner = msg.sender;
134       coffee_address = deliveryaddress;
135     }

137     // Allow the owner to withdraw the money
138     function withdraw() external onlyOwner {
139       owner.transfer(address(this).balance);
140     }

142     // Make this contract destroyable
143     function destroy() external onlyOwner {
144       selfdestruct(owner);
145     }

147     // Allow to delegate the owner
148     function delegateOwner(address who) external onlyOwner {
149       owner = who;
150     }

152     // Adds a Manager
153     function addManager(address who) external onlyOwner {
154       managers[who] = true;
155     }

157     // Removes a Manager
158     function removeManager(address who) external onlyOwner {
159       managers[who] = true;
160     }
161   }

163   // Define the Interface smart contract of the coffee stores,
164   // so we can interact properly with them.
165   contract CoffeeStore is CoffeeData, CoffeeOrder {
166       // Get all coffees inculding details
167       function getCoffeeDB() public view returns (byte[4096], uint8 amount);
168       // Given an id, retrieve the information of the coffee using this id
169       function getCoffee(uint32 id) public view returns (byte[128]);
170       // Get the price of a coffee. Shortcut so one does not have to transfer
171       // lots of data just to get the coffees price.
172       function getPrice(uint32 id) public view returns (uint128);
```

```
174       // Simple ping function, check if a store uses this interface
175       function coffeeStoreAlive() external pure returns (bool);
176       // Order functions for simple orders (1 product, 1 amount, 1 address) and for
177       // comples orders (n products, n amounts, n addresses)
178       function simpleCoffeeOrder(struct_simpleorder) external payable returns (bool);
179       function complexCoffeeOrder(struct_complexorder) external payable returns (bool);
180   }
181
182   // The contract managing the interface for the managers
183   // managers can change some state variables and order manually
184   contract CoffeeManager is CoffeeOwner, CoffeeOrder, LogBook {
185
186     function addCoffeeStore(address storeadr) external onlyManager() {
187       // Test if the address contains the correct interface
188       bool ping = CoffeeStore(storeadr).coffeeStoreAlive.gas(800)();
189
190       // In case of an error: log it
191       if (false == ping) {
192         emit ShopManagement(msg.sender, storeadr, true, false);
193         return;
194       }
195
196       // Iterate through store array and check whether it is already
197       // contained and if not, add it at a proper position
198       uint256 firstfreeslot = 0;
199       uint256 id = 0;
200
201       for (; id < coffee_stores.length; id++) {
202         // Empty store in between
203         if (address(0) == coffee_stores[id]) {
204           firstfreeslot = id;
205         }
206         // store already contained
207         if (storeadr == coffee_stores[id]) {
208           emit ShopManagement(msg.sender, storeadr, true, false);
209           return;
210         }
211       }
212
213       // Add store
214       if (0 == id && 0 == firstfreeslot || 0 != firstfreeslot) {
215         coffee_stores[firstfreeslot] = storeadr;
216         emit ShopManagement(msg.sender, storeadr, true, true);
217         return;
218       }
219
220       assert(coffee_stores.push(storeadr) > 0);
221       emit ShopManagement(msg.sender, storeadr, true, true);
222     }
223
224     // Remove one coffee store from the store list
225     function removeCoffeeStore(address storeadr) external onlyManager() {
226       for (uint256 id = 0; id < coffee_stores.length; id++) {
227         // store found, remove it
228         if (storeadr == coffee_stores[id]) {
229           coffee_stores[id] = 0;
230           emit ShopManagement(msg.sender, storeadr, false, true);
231           return;
232         }
233       }
234
235       // store not found
236       emit ShopManagement(msg.sender, storeadr, false, false);
237     }
238
239     // Set the price for one coffees
240     function setCoffePrice(uint128 price) external onlyManager() {
241       // Set the price and log it
242       emit PriceChanged(msg.sender, coffee_price, price);
243       coffee_price = price;
244     }
```

```
246     // Set the threshold unitl new coffee gets ordered
247     function setThreshold(uint32 threshold) external onlyManager() {
248       // Set the price and log it
249       emit ThresholdChanged(msg.sender, coffee_threshold, threshold);
250       coffee_threshold = threshold;
251     }

253     // Set the address the coffee should get delivered to
254     function setAddress(byte[] newaddress) external onlyManager() {
255       // Set the address and log it
256       emit AddressChanged(msg.sender, coffee_address, newaddress);
257       coffee_address = newaddress;
258     }

260     // A Manager can order coffee manually
261     function manualOrder(address storeadr, struct_simpleorder order)
262       external onlyManager() {
263       // Maybe some assert would be good here in case the amount is too high
264       // Order coffee from the coffee store at storeadr
265       CoffeeStore coffeestore = CoffeeStore(storeadr);
266       uint128 price = coffeestore.getPrice.gas(12000)(order.id);

268       // require(...) should be here to avoid unreasonable prices

270       bool result = coffeestore.simpleCoffeeOrder
271         .gas(27000).value(price*order.amount)(order);

273       // In case of an error: log it, revert changes
274       if (false == result) {
275         emit CoffeeOrderedByManager(storeadr, msg.sender, order.id, order.amount,
276           price*order.amount, false);
277         revert();
278       }

280       // success
281       emit CoffeeOrderedByManager(storeadr, msg.sender, order.id, order.amount,
282         price*order.amount, true);
283     }
284   }


287   // The Contract containing the interface for the users who buy coffee.
288   contract CoffeeContract is CoffeeManager {
289     // Helper function to deserialize byte stream which contains struct_coffee[]
290     function deserializeCoffeeDB(byte[4096] input, uint8 amount) internal pure
291     returns (struct_coffee[]) {
292       struct_coffee[] memory result;
293       uint16 offset = 0;
294       uint16 strlen = 0;

296       for (uint8 c = 0; c < amount; c++) {
297       // Deserialize
298         result[c].id = uint32(uint8(input[offset+0]) +
299           uint8((input[offset+1] << 8)) +
300           uint8((input[offset+2] << 16)) +
301           uint8((input[offset+3] << 24)));

303         result[c].amount = uint32(uint8(input[offset+4]) +
304           uint8((input[offset+5] << 8)) +
305           uint8((input[offset+6] << 16)) +
306           uint8((input[offset+7] << 24)));

308         result[c].price = uint128(uint8(input[offset+8]) +
309           uint8((input[offset+9] << 8)) +
310           uint8((input[offset+10] << 16)) +
311           uint8((input[offset+11] << 24)) +
312           uint8((input[offset+12] << 32)) +
313           uint8((input[offset+13] << 40)) +
314           uint8((input[offset+14] << 48)) +
315           uint8((input[offset+15] << 56)) +
316           uint8((input[offset+16] << 64)) +
317           uint8((input[offset+17] << 72)) +
318           uint8((input[offset+18] << 80)) +
```

```
319             uint8((input[offset+19] << 88)) +
320             uint8((input[offset+20] << 96)) +
321             uint8((input[offset+21] << 104)) +
322             uint8((input[offset+22] << 112)) +
323             uint8((input[offset+23] << 120)));

325           offset += 24;

327           strlen = uint16((input[offset++]));

329           for (uint16 i=0; i < strlen; i++) {
330             result[c].name[i] = input[i+offset];
331           }

333           offset += i;
334         }

336         return result;
337       }

339       // Get coffee databases from every store
340       function getCoffeeSorts() public view returns (struct_coffee_overview) {
341         struct_coffee_overview memory result;
342         byte[4096] memory serialised_result;
343         uint8 counter = 0;
344         uint8 amount = 0;
345         // Iterate through all store addresses and add those plus the coffee database
346         // for this specific store.
347         for (uint256 id = 0; id < coffee_stores.length; id++) {
348           if (address(0) == coffee_stores[id]) {
349             continue;
350           }

352           // add store address
353           result.stores[counter] = coffee_stores[id];

355           // get coffee database and push it
356           (serialised_result, amount) = CoffeeStore(coffee_stores[id]).getCoffeeDB();

358           result.coffeedb[counter] = deserializeCoffeeDB(serialised_result, amount);

360           ++counter;
361         }

363         return result;
364       }

366       // Automatically order coffee
367       function autoOrder(address shop, uint128 orderprice, struct_simpleorder order)
368         private returns (bool) {

370         // invoke the contract of the store and buy some coffee
371         bool result = CoffeeStore(shop).simpleCoffeeOrder
372           .gas(27000).value(orderprice*order.amount)(order);

374         if (result) {
375           emit CoffeeAutomaticallyOrdered(shop, order.id, order.amount, true);
376           return result;
377         }

379         emit CoffeeAutomaticallyOrdered(shop, order.id, order.amount, false);
380         return result;
381       }

383       // This function will be called by a user who buy <quantity> amount of cups of
384       // coffee. This function is very gashungry, have to find a workaround.
385       // One workaround could be that the user who triggeres this order buys his
386       // coffee with gas instead buying it directly with ether
387       function buyCoffee(uint8 quantity, address vote_store,
388         uint32 vote_id) external payable {

390         // Check if enough was paid
391         if (msg.value < quantity*coffee_price) {
```

```
392         emit UserBoughtCoffee(msg.sender, quantity, false);
393         revert();
394       }

396       // add quantity to the number of bought coffees since the last order
397       coffee_count += quantity;

399       // add a vote to the store
400       votes[vote_store][vote_id] += quantity;

402       // increase coffee_count and compare with ThresholdChanged
403       // if coffee_count < threshold then log the successful order and return
404       // otherwise find most voted coffee and automatically order some of it
405       if (coffee_count < coffee_threshold) {
406         emit UserBoughtCoffee(msg.sender, quantity, true);
407         return;
408       } else {
409         // reset coffee_count
410         coffee_count = 0;

412         // order struct
413         struct_simpleorder memory autoorder;
414         autoorder.destination = coffee_address;
415         autoorder.amount = 1;
416         address destination;
417         uint128 orderprice = 0;

419         // Struct for coffee databases of each store
420         struct_coffee[] memory curdb;
421         byte[4096] memory serialised_result;
422         uint8 amount = 0;

424         uint32 maxvotes = 0;

426         // iterate through all stores
427         for (uint256 id = 0; id < coffee_stores.length; id++) {
428           if (address(0) == coffee_stores[id]) {
429             continue;
430           }

432           // Get the coffee database to aquire the ids and the price
433           (serialised_result, amount) =
434               CoffeeStore(coffee_stores[id]).getCoffeeDB.gas(20000)();

436           curdb = deserializeCoffeeDB(serialised_result, amount);

438           // find out if a coffee in this database got more votes than the current
439           // entry in the simpleorder struct and if it can be ordered
440           for (uint32 cid = 0; cid < curdb.length; cid++) {
441             if (votes[coffee_stores[id]][curdb[cid].id] > maxvotes &&
442               curdb[cid].amount > 0) {
443                 // set new max amount of votes and adjust order struct
444                 maxvotes = votes[coffee_stores[id]][curdb[cid].id];
445                 // adjust orderid
446                 autoorder.id = curdb[cid].id;
447                 // adjust price
448                 orderprice = curdb[cid].price;
449                 // set shop address
450                 destination = coffee_stores[id];
451               }

453             // reset votecounter
454             votes[coffee_stores[id]][curdb[cid].id] = 0;
455           }
456         }
457       }

459       // Order coffee and log the results
460       bool res_order = autoOrder(destination, orderprice, autoorder);

462       if (res_order) {
463         emit UserBoughtCoffee(msg.sender, quantity, true);
464         return;
```

```
465       }

467       emit UserBoughtCoffee(msg.sender, quantity, false);
468       revert();
469     }
470   }
```

Listing A.2: Quellcode: coffeecontract.sol

## A.2.2. Solidity Smart Contract - Interface CoffeeStore

```
1   /*
2   Smart Contract: CoffeeStore Interface
3   Purpose: Interface store owner can use to properly communicate with
4            the CoffeeShop Smart Contract
5   Features: Ability to add products, remove product, accept orders
6   Autor: Harald Heckmann
7   */

9   pragma solidity ^0.4.21;

11  // get some extended features including return of structs and nested arrays
12  pragma experimental ABIEncoderV2;

14  contract CoffeeShopData {
15    // Structure which represents one coffee
16    struct struct_coffee {
17      uint32 id;
18      uint32 amount;
19      uint256 price;
20      byte[] name;
21    }
22  }

24  contract CoffeeOrder {
25    // Simple order: One specific product to one specific address
26    struct struct_simpleorder {
27      uint32 id;
28      uint32 amount;
29      byte[] destination;
30    }

32    // Complex order: Multiple products,
33    // each can be delivered to a different address
34    struct struct_complexorder {
35      uint32[] ids;
36      uint32[] amounts;
37      byte[][] destinations;
38    }
39  }

41  // Contracts who want to act as a coffee store and want to be able to
42  // deliver proper interfaces to the CoffeeContract contract, have to implement
43  // this interface
44  contract CoffeeStore is CoffeeShopData, CoffeeOrder {
45    // Get all coffees inculding details
46    function getCoffeeDB() public view returns (byte[4096], uint8 amount);
47    // Given an id, retrieve the information of the coffee using this id
48    function getCoffee(uint32 id) public view returns (byte[128]);
49    // Get the price of a coffee. Shortcut so one does not have to transfer
50    // lots of data just to get the coffees price.
51    function getPrice(uint32 id) public view returns (uint128);

53    // Simple ping function, check if a store uses this interface
54    function coffeeStoreAlive() external pure returns (bool);
55    // Order functions for simple orders (1 product, 1 amount, 1 address) and for
56    // comples orders (n products, n amounts, n addresses)
```

```
57      function simpleCoffeeOrder(struct_simpleorder) external payable returns (bool);
58      function complexCoffeeOrder(struct_complexorder) external payable returns (bool);
59  }
```

Listing A.3: Quellcode: coffeestore.sol

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **BTC** | Bitcoin |
| **DB** | Database |
| **EVM** | Ethereum Virtual Machine |
| **DAO** | Decentralized Autonomous Organization |
| **P2PKH** | Pay-to-Public-Key-Hash |
| **ROM** | Read-Only Memory |
| **UTXO** | Unspent Transaction Output |
| **VM** | Virtual Machine |

# Literaturverzeichnis


[Ant15] Andreas M. Antonopoulos. *Mastering Bitcoin : [unlocking digital cryptocurrencies]*. 1. Aufl. Beijing [u.a.], 2015. ISBN: 9781449374044.

[But13] Vitalki Buterin. *A next generation smart contract & decentralized application platform*. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[Coma] Bitcoin Community. *Dokumentation: Script*. URL: https://bitcoin.org/en/developer-reference#opcodes.

[Comb] Bitcoin Community. *Dokumentation: Standard Transaktionen*. URL: https://bitcoin.org/en/developer-guide#standard-transactions.

[Comc] Ethereum Community. *Dokumentation: Solidity*. URL: https://solidity.readthedocs.io.

[Comd] Ethereum Community. *FAQ: Sharding*. URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQ.

[LRL18] Fujitsu Laboratories Ltd., Fujitsu Research, and Development Center Co. Ltd. *Fujitsu Develops Technology to Verify Blockchain Risks*. Mar. 2018. URL: http://www.fujitsu.com/global/about/resources/news/press-releases/2018/0307-01.html.

[Nak] Satoshi Nakamoto. *Erste Bitcoin Client Versionen*. URL: http://satoshi.nakamotoinstitute.org/code/.

[Nak09a] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: https://www.bitcoin.org/bitcoin.pdf.

[Nak09b] Satoshi Nakamoto. *E-Mail mit erstem Bitcoin Client*. 2009. URL: https://www.mail-archive.com/cryptography@metzdowd.com/msg10152.html.

[Nik+18] Ivica Nikolić et al. *Finding The Greedy, Prodigal, and Suicidal Contracts at Scale*. Tech. rep. School of Computing, NUS Singapore et al., Feb. 2018.

[PB17] Livio Pompianu and Massimo Bartoletti. *An analysis of Bitcoin OP_RETURN metadata*. Apr. 2017.

[Woo14] Gawin Wood. *Ethereum: A secure Decentralised generalised transaction ledger*. 2014. URL: `https://ethereum.github.io/yellowpaper/paper.pdf`.