Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORARBEIT

Blockchain-Technologie im Online Advertising

vorgelegt von

Daniel Braun

Fakultät:	MIN-Fakultät
Studiengang:	B.Sc. Wirtschaftsinformatik
Matrikelnummer:	6922337
Betreuer:	Michael Palk
Erstgutachter:	Prof. Dr. Stefan Voß
	Institut für Wirtschaftsinformatik
Zweitgutachter:	Dr. Kai Brüssau
	Institut für Wirtschaftsinformatik

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

2 Blockchain 1.0 - Wie Bitcoin funktioniert

Mit Fortschritt im Bereich Kryptographie begann auch das Interesse von Forschern an digitalen Währungen. Das Problem dieser frühen Projekte bestand jedoch darin, dass sie einen sogenannten *Central Point of Failure*, also eine zentralisierte Schwachstelle besaßen. Beispielsweise könnten die Konten von Nutzern zwar kryptografisch gesichert, jedoch von zentralen Stellen wie Banken verwaltet werden müssen. Ein wichtiges Problem, welches es mithilfe eines neuartigen Geldsystems zu lösen gilt, ist das sogenannte *Double Spending Problem*. Es muss durch gewisse Mechanismen verhindert werden, dass bösartige Akteure die selben Geldwerte für mehrere Transaktionen verwenden. Bei physischem Geld, also Geldscheinen, Münzen, etc. verhindern komplexe Drucktechniken die Verbreitung von Falschgeld und dadurch dass ein Geldschein nur einmal existieren kann, ist dieser nur für eine Transaktion zu verwenden. Versucht man nun diese Geldwerte gänzlich digital zu verwalten, so liegt die Verantwortung für eine korrekte Beobachtung und Verwaltung bei einer zentralen Stelle wie einer Bank. Diese könnte als Angriffsstelle für Antagonisten dienen und stellt somit eine Gefahr für das System dar. Dieses Kapitel beschäftigt sich mit der traditionellen Funktionsweise von Geld und wie mithilfe eines dezentralen Systems ein zentraler Fehlerpunkt vermieden werden kann.

2.1 Funktionsweise von Geld

Mankiw und Taylor (2018) bezeichnen Geld als ein Bündel von Aktiva, das die Menschen in einer Volkswirtschaft regemäßig dazu verwenden, Waren und Dienstleistungen von anderen Menschen zu erwerben. Es erlaubt den Parteien einem Tauschgeschäft, bei dem beide Seiten mit dem Gut des Tauschpartners zufrieden sein müssen, zu entgehen und ermöglicht stattdessen eine effiziente Allokation von Ressourcen. Zugleich stellt Geld sicher, dass das eigene Kapital den Wert auch in Zukunft behält. Damit ein Handelsgut als Geld angesehen werden kann, muss es drei Funktionen erfüllen können. Fundamental ist, dass das Handelsgut generell als Tausch- bzw. Zahlungsmittel akzeptiert wird. Theoretisch könnte man versuchen, sein Abendessen mit dem eigenen Fahrrad zu bezahlen, doch kommt man in der Praxis mit dieser Strategie nicht weit. Des Weiteren muss das Tauschgeschäft als Recheneinheit fungieren können. Dies ist notwendig, da anhand dessen die relativen Preise anderer Waren in der Marktwirtschaft ermittelt werden müssen. Zuletzt muss sichergestellt sein, dass das Handelsgut wie bereits erwähnt in Zukunft auch seine Kaufkraft behält. Jemand, der es als Zahlungsmittel akzeptiert, muss sich darauf verlassen können, dass es auch für zukünftige Geschäfte verwendet werden kann.

Bei den Geldformen unterscheidet man zwischen Warengeld und Rechengeld. Diese unterscheiden sich in ihrem intrinsischen Wert, also darin, ob sie auch außerhalb von Tauschgeschäften einen Nutzen finden. Ein Beispiel für Warengeld ist Gold, welches neben Tauschgeschäften auch industriell verarbeitet werden kann. Papiergeld hingegen bietet abseits des Tauschgeschäftes keinen Nutzen für den Besitzer. Um trotzdem den Wert des Geldes gewährleisten zu können, wird es von Seiten des Staats als universelles Zahlungsmittel in der jeweiligen Marktwirtschaft bestimmt.

Eine weitere wichtige Rolle im Finanzsystem nehmen Zentralbanken ein. Sie überwachen das Bankensystem und steuern über eine geeignete Geldpolitik das Geldangebot auf dem Markt. Durch das Drucken von Geld und den anschließenden Kauf von Wertpapieren können sie das Geldangebot erhöhen. Um es wiederum zu verringern, verkaufen sie Wertpapiere und nehmen das erhaltene Geld aus dem Umlauf.

Eine Währung, die zum Verwalten und Versenden von monetärem Wert dient, hat drei technische Anforderungen zu erfüllen:

1. Sicherstellung des Wertes, also die Authentizität
2. Garantie dafür, dass die selbe Währung nicht mehr als einmal verwendet werden kann (Double Spending)
3. Zugang zur Währung nur für befugten Besitzer

TODO

2.2 Notwendigkeit für Blockchain-Technologie

2.3 Theorie der Blockchain-Technologie am Beispiel von Bitcoin

Auch wenn es andere Projekte für dezentrale Währungen wie B-Money [Dai (1998)] und Hashcash [Back (2002)] gab, begann der Aufschwung digitaler Währungen im Jahr 2008 mit der Veröffentlichung des Bitcoin-Whitepapers [Nakamoto (2008)]. Diese Publikation trägt den Titel *Bitcoin: A Peer to peer Electronic Cash System*, wurde von einer bis heute unbekannten Person, unter dem Namen *Satoshi Nakamoto* veröffentlicht und kombinierte Technologien ihrer Vorgänger. Statt einer zentralen Verwaltungsstelle handelt es sich bei Bitcoin um ein dezentrales Peer-to-peer Netzwerk zwischen den Nutzern des Bitcoin-Protokolls. Außerdem werden Vermögenswerte nicht durch klassische Münzen auf einem Konto repräsentiert, sondern durch vergangene Transaktionen in einem dezentralen und öffentlichen Transaktionsbuch, dem sogenannten *Ledger*, impliziert. Aufgrund dieser Ei-

genschaften besteht keine zentrale Angriffsfläche für bösartige Akteure und jeder Akteur im Netzwerk hat Kenntnis über alle Transaktionen. Die folgenden Untersektionen beschäftigen sich mit der Verwaltung und dem Zugang für Nutzer, die Funktionsweise von Transaktionen sowie die Art und Weise, wie die verschiedenen Akteure im Netzwerk zu einem gemeinsamen Konsens kommen.

2.3.1 Keys und Adressen

Als Kryptographie bezeichnet man Verfahren zur Verschlüsselung von Informationen, die schon von den Nazis im zweiten Weltkrieg genutzt wurden [Landwehr (2008): 19]. Mithilfe von Maschinen, den sogenannten *ENIGMA*, verschlüsselten sie wichtige strategische Informationen wie die Aufenthaltsorte von Truppen oder taktische Befehle, die anschließend per Funk überbracht wurden. Kryptographische Verfahren folgten zu der Zeit dem Prinzip *Security by Obscurity*, nach dem die Sicherheit eines Verschlüsselungsverfahrens davon abhängig ist, ob die Funktionsweise dieser bekannt ist. Dies hatte zur Folge, dass im Falle der Nazis, deren ENIGMA-Code durch die Anstrengungen polnischer und später englischer Wissenschaftler um Alan Turing unschädlich gemacht werden konnte [Landwehr (2008): 18/19].

Im Jahr 1976 stellten *Diffie* und *Hellman* die bis dahin unbekannte asymmetrische Verschlüsselung vor, bei der jede Partei ein Schlüsselpaar, bestehend aus privatem und öffentlichem Schlüssel, besitzt [vgl. Diffie und Hellman (1976)]. Derartige Verfahren sind heutzutage der Standard und werden auch im Bitcoin-System verwendet. Das restliche, sowie folgende Kapitel beziehen sich auf [Antonopoulos (2014)] und ziehen, wo es nötig ist, zusätzliche Quellen hinzu.

Für Bitcoin wird ein Paar aus Schlüsseln erzeugt. Dieses Paar besteht aus dem privaten Schlüssel (Private Key), welcher nur dem Besitzer bekannt ist und zum Signieren von Transaktionen nötig ist. Aus diesem wird durch die Verwendung von Hashing-Verfahren ein öffentlicher Schlüssel (Public key) abgeleitet, mit dem Bitcoins empfangen werden können. Außerdem kann aufgrund der mathematischen Abhängigkeit zwischen den Schlüsseln eine durch den privaten Schlüssel signierte Transaktion mithilfe des öffentlichen Schlüssels verifiziert werden. Dies geschieht, indem der Absender die Transaktion mit seinem privaten Schlüssel signiert und die Authentizität der Signatur mithilfe des öffentlichen Schlüssels von anderen Akteuren des Netzwerks verifiziert wird. Um Begünstigter einer Transaktion zu sein, muss man eine Adresse besitzen und diese an andere Nutzer des Netzwerks propagieren. Um jene zu erzeugen, wird der öffentliche Schlüssel genutzt,

welchen man nicht wieder aus der Adresse rekonstruieren kann.

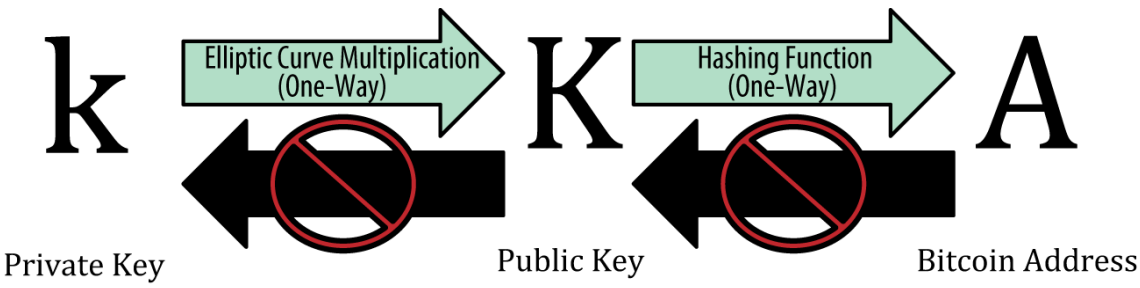

Abbildung 2.1.: Generierung der Schlüssel bzw. Adressen aus dem jeweiligen Vorgänger. Quelle: Antonopoulos (2014)

Private Keys

Ein privater Schlüssel besteht aus einer Zahl von 256 zufälligen Bits. Er wird zum Signieren von Transaktionen und für den Zugriff auf ein Guthaben benötigt. Ohne privaten Schlüssel verliert man als Besitzer von Bitcoin auch den Zugriff auf das eigene Guthaben. Um einen privaten Schlüssel generieren zu können, benötigt man eine sichere Quelle für Zufälligkeit. In anderen Worten: Die Wahl der zufälligen Zahl darf nicht vorhersehbar sein. Dazu verwendet die Bitcoin-Software den Random Number Generator des verwendeten Betriebssystems, kombiniert mit einem menschlichen Input, wie dem Bewegen der Maus [vgl. Antonopoulos (2014): 58]. Mithilfe des Generators erzeugt man einen zufälligen String, welcher *mehr* als 256 Bits hat. Diesen lässt man anschließend durch den SHA256 Hash-Algorithmus laufen und prüft, ob die resultierende Zahl kleiner ist, als die vom Bitcoin-Protokol gewählte Konstante n ($n = 1.1578 * 10^{77}$). Diese Zahl entspricht nach der Ordnung der Gruppe, die in der dort verwendeten elliptischen Kurve entspricht. Nach [Corbellini (2015)] befinden sich alle Punkte der Kurve in dieser Gruppe, weshalb der gewählte private Schlüssel, welcher für die anschließende Generierung eines öffentlichen Schlüssels genutzt wird, nicht außerhalb dieses Bereichs liegen darf.

Public Keys

Um einen öffentlichen Schlüssel aus dem privaten generieren zu können, benötigt man ein kryptografisches Verfahren, welches eine Rekonstruktion des Privaten aus dem öffentlichen Schlüssel nicht zulässt. Das vom Bitcoin-Protokol verwendete Verfahren wird *Elliptic Curve Cryptography* gennant und bedient sich an den Eigenschaften einer Ellipse. Um einen öffentlichen Schlüssel zu generieren, wählt man einen Punkt, den sogenannten

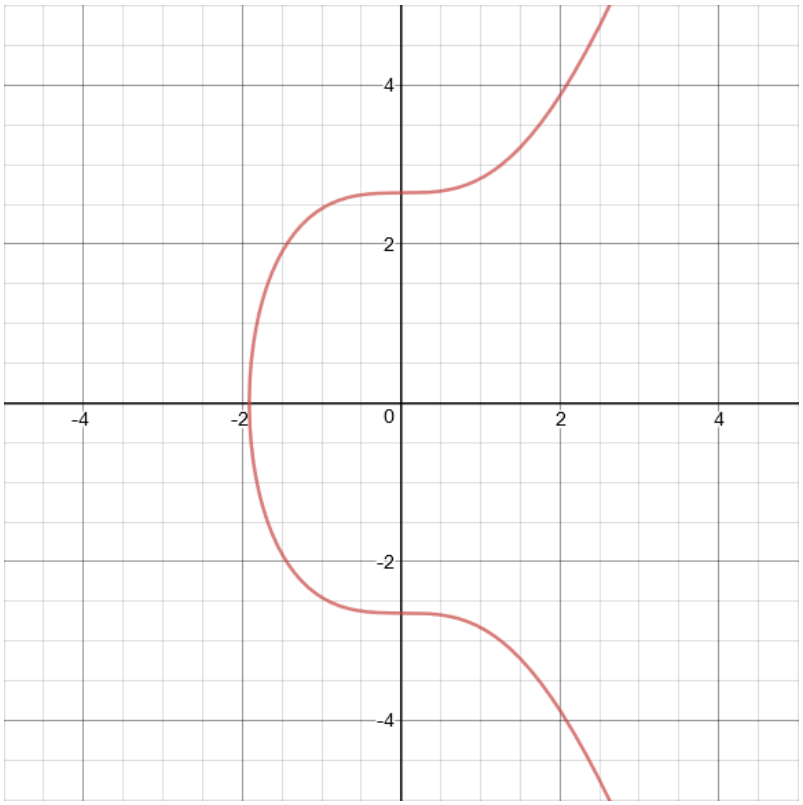

Abbildung 2.2.: Die von Bitcoin verwendete Ellipse mit der Funktion $y^2 = x^3 + 7$
TODO: Bessere Grafik

Generatorpunkt, auf der Ellipse und multipliziert diesen mit dem vorher generierten privaten Schlüssel. Eine Multiplikation kann auch als Addition einer Zahl mit derselben betrachtet werden. Um den Punkt G auf der Ellipse mit sich selbst zu addieren, zieht man an diesem die Tangente und berechnet den Schnittpunkt von Ellipse und der gezogenen Tangente. Anschließend spiegelt man den Punkt an der x-Achse und erhält somit 2G. Diese Addition führt man so oft aus, wie der 256 Bit lange private Schlüssel groß ist, sodass man am Ende einen Punkt (x,y) erhält, welcher als öffentlicher Schlüssel genutzt werden kann. Diesen generierten Schlüssel kann man veröffentlichen, denn aus ihm lässt sich nicht schließen, mit welchem Faktor der Generatorpunkt multipliziert wurde.

Bitcoin Adressen

Eine Adresse ist ein aus dem öffentlichen Schlüssel generierter String aus Buchstaben und Zahlen, der den Besitzer des Schlüssels zu einem potentiellen Empfänger einer Transaktion macht. Beim diesem muss es sich allerdings nicht zwangsläufig um eine Person handeln, denn auch Organisationen, geschriebene Skripte, etc. kommen als *abstrakter* Empfänger in Frage. So wie der öffentliche aus dem privaten Schlüssel erzeugt wird, wird die Bitcoin Adresse aus dem öffentlichen Schlüssel mithilfe von Hashing-Algorithmen er-

zeugt. Die verwendeten Algorithmen, welche nacheinander auf den öffentlichen Schlüssel angewendet werden, heißen SHA-256 und RIPEMD160. Da Verschlüsselungsalgorithmen einen Grundbaustein für Blockchain-Technologie darstellen, wird ihre Funktionsweise im Folgenden beispielhaft anhand des SHA-256-Algorithmus erläutert.

2.3.2 Einschub: SHA-256

Ein Hashalgorithmus ist eine mathematische Funktion, die einen Input entgegennimmt und einen Output fester Größe, den sogenannten Hashwert, erzeugt. Ein Hashalgorithmus sollte idealerweise folgende Eigenschaften bieten:

- Für einen gegebenen Input immer denselben Output generieren
- Nur in eine Richtung berechenbar sein
- Durch geringe Änderungen am Input einen völlig anderen Output generieren

Hashfunktionen haben vielerlei Anwendungsmöglichkeiten im Bereich der Datensicherheit und eine verwendete Hashfunktion im Bitcoin-Protokoll ist die bereits erwähnte SHA256-Funktion, welche u.A. von der NSA entwickelt wurde. Das folgende Unterkapitel orientiert sich an Dang (2015), ist allerdings für das Verständnis nachfolgender Kapitel nicht zwingend erforderlich.

Definitionen

Bevor der Algorithmus erläutert werden kann, ist es notwendig folgende Operationen, sowie benötigte Konstanten zu definieren. Die Operationen werden auf Zahlen in Binärform bitweise angewendet.

Tabelle 2.1.: Benötigte Operationen

+	Addition in Mod 2^{32}
XOR	Vergleichender Operator $\oplus$. Ergebnis ist True(1), wenn genau einer der beiden Inputs True ist. Andernfalls wird False(0) ausgegeben. Beispiel: $0 \oplus 1 = 1$

Rotation Right	Verschiebung der Bits um n Stellen nach rechts. Bei Overflow werden Bits wieder vorne angefügt. Beispiel: $$ROTR_1(011) = 101$$
Shift Right	Ähnlich wie die Rotation, nur dass Bits an letzter Stelle wegfallen und eine Nullen vorne angefügt werden. Beispiel: $$SHR_1(011) = 001$$
Choice	Nimmt drei Zahlen gleicher Länge entgegen und entscheidet anhand der Bits des ersten Parameters, welches Bit der jeweils anderen Parameter übernommen werden soll. $$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$ Beispiel: $Ch(110,001,101) = 001$
Majority	Nimmt drei Zahlen gleicher Länge entgegen und übernimmt an jedem Bit denjenigen Wert, der zwischen den Inputs am häufigsten auftaucht. $$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$ Beispiel $Maj(110,001,101) = 101$
σ_0	σ_0 und die folgenden Funktionen sind definierte Folgen der oben definierten Operationen. $$\sigma_0(x) = ROTR_7(x) \oplus ROTR_{18}(x) \oplus SHR_3(x)$$
σ_1	$$\sigma_1(x) = ROTR_{17}(x) \oplus ROTR_{19}(x) \oplus SHR_10(x)$$

Σ_0

$$\Sigma_0(x) = ROTR_2(x) \oplus ROTR_{13}(x) \oplus ROTR_{22}(x)$$

Σ_1

$$\Sigma_1(x) = ROTR_6(x) \oplus ROTR_{11}(x) \oplus ROTR_{25}(x)$$

Außerdem müssen noch zwei Listen mit Konstanten definiert werden. Die Liste K beinhaltet die ersten 32 Bit der Nachkommastellen der Kubikwurzeln der ersten 64 Primzahlen. H die ersten 32 Bit der Nachkommastellen der Quadratwurzeln der ersten 8 Primzahlen. Die genauen Werte haben keine sonderliche Bedeutung, sondern lediglich die scheinbare Zufälligkeit ist von Relevanz. Indem man diese berechenbaren Zahlen nimmt, minimiert sich die Wahrscheinlichkeit für eine absichtlich platzierte Schwachstelle im Algorithmus.

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4
ab1c5ed5 d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe
9bdc06a7 c19bf174 e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f
4a7484aa 5cb0a9dc 76f988da 983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967 27b70a85 2e1b2138 4d2c6dfc
53380d13 650a7354 766a0abb 81c2c92e 92722c85 a2bfe8a1 a81a664b
c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070 19a4c116
1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7
c67178f2
```

Listing 2.1: Liste K von Konstanten

```
H0 = 6a09e667
H1 = bb67ae85
H2 = 3c6ef372
H3 = a54ff53a
H4 = 510e527f
H5 = 9b05688c
H6 = 1f83d9ab
H7 = 5be0cd19
```

Listing 2.2: Liste H mit den Arbeitsvariablen H0 - H7

Die Zahlen befinden sich zu diesem Zeitpunkt in Hexadezimal-Form, müssen aber in Binärzahlen umgewandelt werden.

Vorbereitung der Nachricht

Bevor eine Nachricht verarbeitet werden kann, muss sie in eine für den Algorithmus brauchbare Form gebracht werden. Da der SHA-256 mit Zahlen in Binärform arbeitet, müssen Inputs in Stringformat über die ASCII-Tabelle in Binärzahlen umgewandelt werden. So wird z.B. aus dem String 'ab' die Zahl 01100001 01100010. Anschließend fügt man eine Eins und k-viele Nullen hinzu, dass ein Vielfaches von 512 abgezogen 64 herauskommt. Dang (2015) definiert diese Voraussetzung durch

$$l + 1 + k \equiv 448 Mod 512$$

mit l als Länge der Nachricht. Anschließend werden 64 Bits hinzugefügt, welche die Zahl l repräsentieren, im Fall 'ab' wäre das die 16 in 64-Bit Repräsentation. Die resultierende N*512-Bit Zahl stellt den Input für die eigentliche Verschlüsselung dar.

Aufteilung der Nachricht in Blöcke

Die resultierende Zahl wird anschließend in Nachrichtenblöcke M_1 - M_N der Länge 512-Bit und diese wiederum in 16 32-Bit-Blöcke M_{i0} - M_{i15} aufgeteilt, die man auch Wörter nennt. Der String 'ab' resultiert in einer 512-Bit-Zahl, sodass an dieser Stelle nur eine Aufteilung in 16 32-Bit-Blöcke nötig wäre.

Algorithmus

Im folgenden soll der Algorithmus anhand eines Pseudo-Codes erläutert werden. Zu beachten ist, dass die Liste k, sowie die Variablen H0-H7 global definiert seien.

```
1  For i=1 to N:
2    w = []
3    For t=0 to 63:
4      if t <= 15:
5        w.push(M[i][t])
6      else:
7        a = σ1(w[t-2]) + w[t-7] + σ0(w[t-15]) + w[t-16]
8        w.push(a)
9    a = H0
10   b = H1
11   c = H2
12   ...
```

```
13      h = H7
14      For j=0 to 63
15        T1 = h + Σ1(e) + Ch(e,f,g) + k[j] + w[j]
16        T2 = Σ0(a) + Maj(a,b,c)
17        h = g
18        g = f
19        e = d + T1
20        d = c
21        c = b
22        b = a
23        a = T1 + T2
24      H0 = a + H0
25      H1 = b + H1
26      ...
27      H7 = h + H7
28    result = ''.concat(H1).concat(H2). ... .concat(H7)
```

Listing 2.3: Pseudocode zu SHA256

Es wird zu Beginn ein leerer String initialisiert, der am Ende das Ergebnis des Algorithmus darstellt [siehe Zeile 1 im Code]. Anschließend wird die erste Schleife gestartet, die für jeden der N 512-Bit-Blöcke durchlaufen wird. Zu Beginn jedes Schleifendurchlaufs wird eine leere Liste *w* initialisiert, die während des Durchlaufs gefüllt wird [2-3]. Für das Füllen der Liste wird eine weitere Schleife initialisiert, die von 0 bis 63 läuft und in den ersten 16 Durchgängen lediglich die 16 Wörter aus M[i] in die bisher leere Liste w einfügt [4-6]. Ab Schleifendurchgang 16 werden die übrigen Einträge mithilfe der vorher definierten Funktionen berechnet, sodass man am Ende eine Liste mit 64 Einträgen hat [7-9]. Anschließend werden die sogenannten Arbeitsvariablen a bis h mithilfe der vorher definierten H0-H7 initialisiert [10-14]. Die folgende Schleife ist die, in der die eigentliche Kompression stattfindet. Sie läuft erneut von 0 bis 63 [15]. Mithilfe der vorher Definierten Funktionen und der Arbeitsvariablen werden die temporären Variablen T1 und T2 ermittelt. Für T1 spielen zusätzlich die Listen k, welche die Kubikzahlen der ersten 64 Primzahlen enthält, und die Liste w, welche die in Zeile 4 bis 9 generierten Wörter enthält, eine Rolle [16-17]. Anschließend werden die Arbeitsvariablen mithilfe der Temporären neu gesetzt, sodass man nach 64 Durchläufen scheinbar willkürliche Zahlen erhält [18-23]. Man addiert schließlich die Arbeitsvariablen und ursprünglichen Variablen H0-H7 und kettet diese aneinander [24-28]. Bei einer kleinen Nachricht wie dem 'ab', welche lediglich einen 512-Bit-Block benötigt, wäre dies bereits das Ergebnis des Algorithmus. Sollte der Input ein Vielfaches von 512-Bit benötigen, wird die äußerste Schleife mehrfach durchlaufen und die Variablen H0-H7 in den Zeilen 24-28 nicht mehr mithilfe der

initial definierten H0-H7, sondern mit denen der vorangegangenen Iteration berechnet. Unabhängig von der Länge des Inputs erhält man so immer einen Output von 256 Bits, da die Arbeitsvariablen jeweils eine Länge von 32 Bits haben.

2.3.3 Wallet

Eine Wallet ist ein Programm, welches als Interface zwischen Bitcoin-Netzwerk und dem Nutzer dient. Dessen Funktionen beinhalten die Verwaltung der Schlüssel, das Berechnen des Guthabens und das Signieren von Transaktionen. Eine Wallet ist, im Gegensatz zu einer physikalischen Geldbörse, nicht für das Halten von Münzen, sondern zur Verwaltung der privaten Schlüssel zuständig. Wie das Berechnen des 'Guthabens' geschieht, wird im Unterkapitel *Transaktionen* erläutert.

Man unterscheidet zwischen nicht-deterministischen und deterministischen Wallets. Die erste Variante kann man sich als Korb vorstellen, in dem vorher zufällig generierte private Schlüssel in großer Anzahl gelagert sind. Dabei erzeugt ein privater einen öffentlichen Schlüssel, der wiederum eine Adresse erzeugt (siehe Kapitel *Keys und Adressen*). Um die eigene Pseudonymität zu schützen, ist es empfehlenswert, einen Key nur ein Mal zu benutzen. Aufgrund der hohen Anzahl angesammelter Keys und der damit verbundenen Datensicherung ist diese Art Wallet heute nicht mehr der Standard.

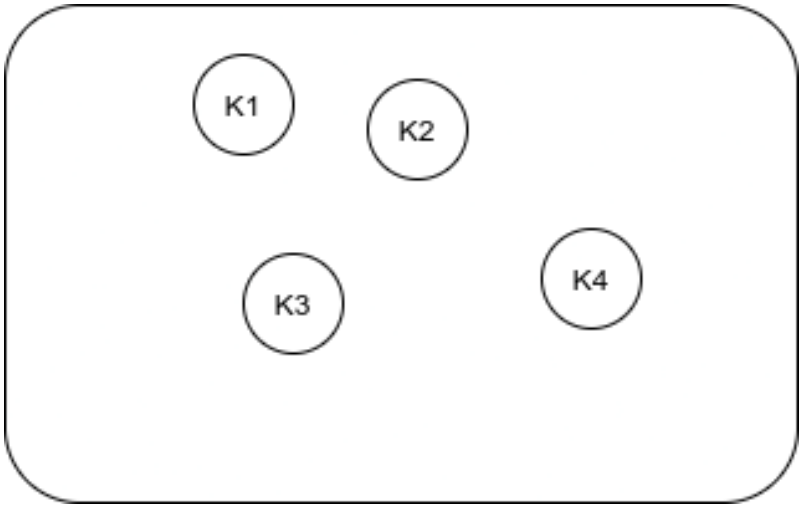

Abbildung 2.3.: Nicht-deterministische Wallet
Quelle: Eigene Darstellung

Die fortgeschrittenste Form einer deterministischen ist die sogenannte BIP32-Wallet, welche zwei nützliche Eigenschaften aufweisen kann (BIP steht für *Bitcoin Improvement Proposal* und bezeichnet eine nachträgliche Ergänzung zum Bitcoin-Ökosystem) [Antonopoulos (2014)]. Diese werden in Buterin (2013a) als *Master Public Key Property* und

Hierarchy Property bezeichnet. Die Master Public Key Property beschreibt die Möglichkeit, aus einem Master Private einen Master Public Key zu generieren, der wiederum alle öffentlichen Schlüssel und deren Adressen erzeugen kann. Dazu berechnet man den sogenannten Offsets, indem man den gewünschten Index und den Master Public Key addiert und das Ergebnis als Input für eine Hashfunktion verwendet. Anschließend addiert man Offset und Master Public Key und erhält den öffentlichen Schlüssel am Index.

$$offset = SHA256(index + masterPubKey)$$

$$pubKey_{index} = offset + masterPubKey$$

Dies geht analog genauso mit dem Master Private Key. Aufgrund dieser Eigenschaft ist es möglich, den Master Public Key ungeschützt zu lagern und sogar an dritte Parteien herauszugeben, ohne dass diese Zugriff auf das Guthaben erhalten.

Die Hierarchieeigenschaft wird im Kontext einer Organisation mit verschiedenen Organisationszweigen interessant. Ein Geschäftsführer könnte so den unterschiedlichen Geschäftszweigen seines Unternehmens Schlüsselpaare zuweisen, wodurch diese die Verfügungsgewalt über das eigene und Guthaben von Unterstellen erhalten. Gleichzeitig behält der Geschäftsführer die absolute Kontrolle über alle Schlüssel, da er im Besitz der Master Keys ist. Anders als bei einer nicht-deterministischen Wallet müssen nicht mehr die Priva-

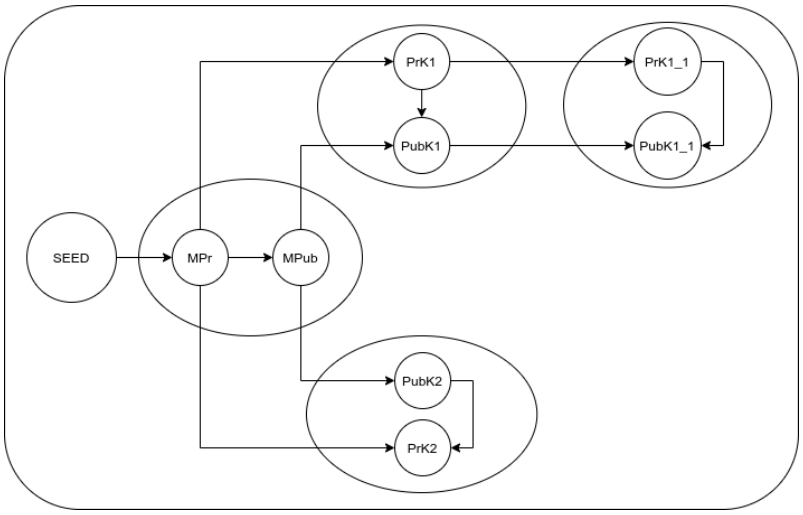

Abbildung 2.4.: BIP32-Wallet
Quelle: Eigene Darstellung

te Keys selbst, sondern lediglich der *Seed* gesichert werden. Dieser ist seit dem BIP39/44 eine kurze Liste von für den Menschen leserlichen Worten. Diese können mithilfe eines speziellen Dictionaries in Hex-Zahlen umgewandelt werden, aus denen schließlich der Master Private Key erzeugt wird.

2.3.4 Transaktionen

Damit Bitcoin die Funktion des Zahlungs- bzw. Tauschmittels erfüllen kann, müssen Transaktionen vom Netzwerk ermöglicht werden. In einem *Distributed Ledger*, einer Art dezentraler Datenbank, werden alle Transaktionen im Netzwerk aufgezeichnet. Da es sich um ein gänzlich digitales und dezentrales Netzwerk handelt, sind spezielle Datenstrukturen und Mechanismen nötig.

Transaction Outputs

Das Guthaben einer Wallet befindet sich nicht an einem Ort, sondern wird aus *Unspent Transaction Outputs*, kurz UTXO zusammengesetzt, welche alle verfügbaren Gutschriften aus vergangenen Transaktionen darstellen. Eine Wallet scannt die Blockchain nach allen Outputs, welche mit den Keys des Besitzers assoziiert sind und errechnet damit das gesamte verfügbare Guthaben. Außerdem speichert sie die nötigen Referenzen für den Zugriff auf die UTXO für den späteren Gebrauch in einer Datenbank ab. UTXO sind diskrete, sowie unteilbare Einheiten und können demnach nur in ihrer Gesamtheit genutzt werden. Man betrachte die folgende Abbildung 2.5:

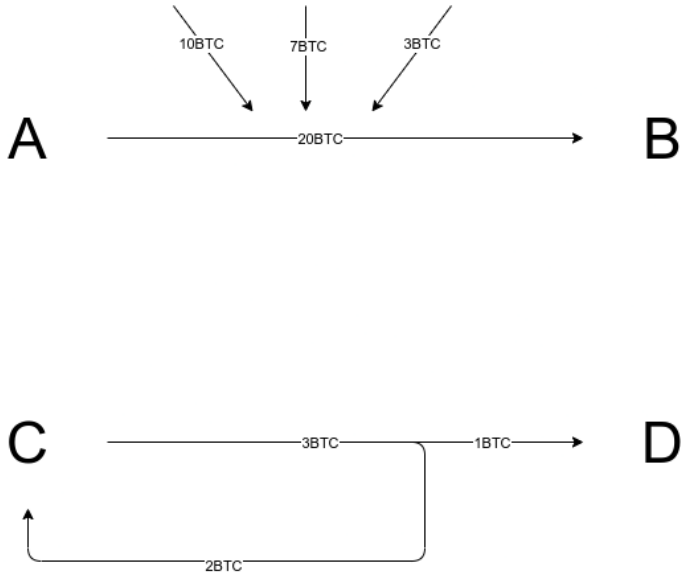

Abbildung 2.5.: Transaktionen mit unteilbaren UTXO
Quelle: Eigene Darstellung

Für die gewünschte Transaktion von A nach B werden 20 BTC benötigt, die von der Wallet aus verfügbaren UTXO zusammengefügt werden. Für das optimale Zusammensammeln von Outputs ist die Wallet zuständig. C hat für die gewünschte Transaktion von 1 BTC nur 3 BTC zur Verfügung, weshalb diese in ihrer Gesamtheit aufgebraucht werden müssen. Die übrigen 2 BTC gehen jedoch nicht verloren, sondern landen lediglich wieder

als Wechselgeld bei C. Die gewünschte Transaktion hat also 2 Outputs. Jeder Output besteht aus 2 Komponenten: Der Anzahl von BTC und einem sogenannten *Locking Script*, welches nur mithilfe des assoziierten private Keys gelöst werden kann.

Transaction Inputs

Das Gegenstück zu den Outputs sind Inputs, über welche die benötigten UTXO für eine Transaktion gesammelt werden. Sie bestehen aus:

- Einer ID, über die eine bestimmte Transaktion referenziert wird
- Vout, einem Index über den auf bestimmte UTXO der zuvor referenzierten Transaktion zugegriffen wird
- Dem Unlocking Script, welches das Locking Script des ausgewählten UTXO löst und mithilfe des privaten Schlüssels erzeugt wird
- Einer Sequence, einem optionalen Feld zur Sperrung der Outputs für eine bestimmte Zeit

Sowohl Input, als auch Output werden serialisiert und als Byte-Streams im Netzwerk propagiert.

Transaktionskosten

Abhängig von der Auslastung des Netzwerkes wird eine dadurch bestimmte Menge an Transaktionsgebühren nötig, die sowohl als Anreiz für Bitcoin-Miner, als auch Abschreckungsmechanismus gegen Angriffe durch hochfrequentes Senden von kleinen Transaktionen dienen. Diese sind zwar kein Muss, jedoch bevorzugen die Miner Transaktionen mit höheren Transaktionskosten. Das Berechnen von optimalen Transaktionsgebühren wird normalerweise von der Wallet ausgeführt, jedoch können auch manuelle Transaktionen erstellt werden, die ohne Transaktionskosten langsamer oder gar nicht vom Netzwerk verarbeitet werden. Bei einer manuellen Transaktion müssen sowohl Inputs, als auch Outputs definiert werden und die Transaktionsgebühren ergeben sich implizit aus der Differenz jener Komponenten. So wie man keine Transaktionsgebühren angegeben kann, ist auch das versehentliche Angeben von zu hohen Gebühren möglich. Man betrachte erneut die Abbildung 2.5, in der C 3 BTC an D sendet. Die Transaktion hat 2 Outputs, 1 BTC an D und 2 BTC zurück an C. Sollte der Sender vergessen den 2. Output, welcher das Wechselgeld darstellt, zu definieren, dann werden die gesamten 2 BTC als Transaktionsgebühren für die Transaktion genutzt. Zusammenfassend lassen sich Transaktionen als

Sammlungen von Inputs und Outputs definieren, deren Differenz die genutzten Transaktionskosten darstellen.

2.3.5 Die verschiedenen Akteure im Netzwerk

In **?** nennt Nakamoto das Propagieren von Transaktionen an alle Knoten als ersten Schritt, der zur Instandhaltung des Netzwerkes erforderlich ist. Dieses Kapitel steigt einen Schritt früher ein und befasst sich mit den verschiedenen Akteuren im dezentralen Netzwerk, ihren Rollen sowie den nötigen Schritten zum Eintritt neuer Teilnehmer.

Rollen

Trotz fehlender Hierarchien im Netzwerk nehmen die Teilnehmer abhängig von ihren Funktionalitäten verschiedene Rollen ein. Als *Full Node* bezeichnet man Teilnehmer, die alle der folgenden Funktionalitäten erfüllen:

- Routing Modul - Kommunikation mit dem Netzwerk
- Wallet - Verwaltung von Guthaben
- Miner - Finden neuer Blöcke
- Full Blockchain - Besitz gesamter Blockchain

Diese können selbstständig Transaktionen prüfen, signieren und am Konsens-Algorithmus teilnehmen.

Bis auf das Routing-Modul, ohne das die Kommunikation mit dem Netzwerk nicht möglich ist, können sich die Teilnehmer ein Subset der Funktionalitäten heraussuchen und diese erfüllen. Beispielsweise benötigt eine Lightweight-Wallet kein Mining und auch keine volle Blockchain. Stattdessen lässt sie Transaktionen von einer dritten Partei prüfen und könnte so aufgrund des geringen Speicherbedarfs in einem Internetbrowser laufen.

Bootstrapping neuen Teilnehmers

Nach dem Hochfahren muss ein neuer Teilnehmer zunächst mindestens einen anderen im Netzwerk finden. Dabei stehen ihm sogenannte *DNS Seeds* zur Verfügung, die nichts anderes sind als Server, die auf Abfrage Adressen von stabil laufenden Teilnehmern zurückgeben und deren Adressen im Bitcoin-Klienten enthalten sind. Alternativ kann ein neuer Knoten auch eine manuelle Verbindung zu einem anderen Teilnehmer eingehen, wenn ihm dessen Adresse bekannt ist. Sobald ein anderer Teilnehmer gefunden wurde,

wird über TCP ein *Handshake* ausgeführt: Der Neue sendet seine Informationen, wie die Version seines Bitcoin-Klienten an den gefundenen Teilnehmer und dieser geht bei Validität der Informationen eine Verbindung mit ihm ein. Anschließend können sie ihnen bekannte Adressen austauschen, dieser Vorgang wird in Abbildung 2.6 veranschaulicht:

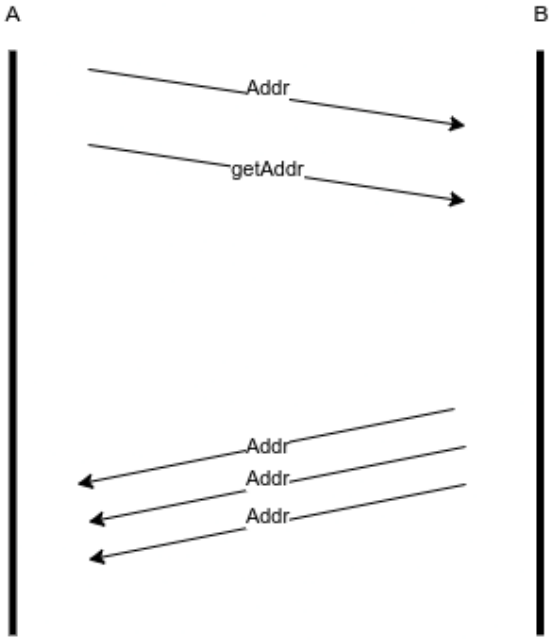

Abbildung 2.6.: Teilnehmer tauschen nach erfolgreicher Verbindung Adressen aus
Quelle: vgl. Antonopoulos (2014)

Ebenso wie sie Adressen miteinander austauschen, werden auch Informationen über den Zustand der Blockchain ausgetauscht. Sollten dem neuen Teilnehmer Blöcke fehlen, versucht er diese von seinen neuen Nachbarn anzufordern, um seine eigene Blockchain zu synchronisieren. Dabei wird der Aufwand auf die verschiedenen Nachbarn aufgeteilt, um einzelne Teilnehmer nicht zu überlasten. Es ist dasselbe Prinzip, wie beim schon länger praktizierten File-Sharing.

2.3.6 Blockchain

Die Blockchain ist die Kerntechnologie des gesamten Protokolls und stellt im Kern eine Datenstruktur mit einer verketteten Liste von Blöcken dar, die wiederum Transaktionen enthalten. Die Verkettung erfolgt dadurch, dass jeder Block einen vorher erzeugten referenziert. Genauer gesagt enthält der neue Block den SHA256-Hash des Vorgängerblocks, mit dem dieser eindeutig identifizierbar ist. Dadurch wird der Hashwert des Nachfolger verändert und dies wird so für alle folgenden Generationen fortgesetzt. Die folgende Abbildung 2.7. zeigt, wie der zweite Block den Vorgängerblock mithilfe des Hash-Werts referenziert.

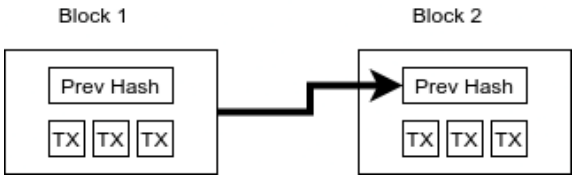

Abbildung 2.7.: Neuer Block referenziert alten Block
Quelle: Eigene Darstellung

Der daraus entstehende Vorteil ist, dass ein Angreifer bei Veränderung eines vergangenen Blocks die Hashwerte aller Nachfolger neu berechnen muss. Wieso dies mit sehr hohem Rechenaufwand verbunden ist, wird später näher erläutert.

Datenstruktur

Die Datenstruktur des Blocks besteht aus mehr Komponenten als dem Vorgänger-Hash und den Transaktionen:

```
1  Block {
2    Block Size;
3    Block Header;
4    Transaction Counter;
5    Transactions;
6  }
7
8  Block Header {
9    Version;
10   Prev Block Hash;
11   Merkle Root;
12   Timestamp;
13   Difficulty Target;
14   Nonce;
15 }
```

Listing 2.4: Datenstruktur des Blocks

Der zuvor angesprochene Hashwert wird nicht aus dem gesamten Block, sondern dem *Block Header* errechnet, der Meta-Daten zum Block enthält. Die Komponenten *Merkle Root*, *Difficulty Target* und *Nonce* werden an geeigneter Stelle erläutert.

Abgesehen vom Block-Hash lässt sich ein Block auch über die Block-Höhe, was lediglich

dem Index des Blocks in der Liste entspricht, identifizieren. Dafür folgt man vom Block aus über die Vorgänger der Blockchain, bis man beim sogenannten *Genesis Block*, dem ersten Block, ankommt. Anschließend zählt man den Abstand und erhält die Block-Höhe. Eine eindeutige Identifikation des Blocks ist hierüber allerdings nicht möglich, denn es kann im Falle von *Forks*, die im Kapitel *Der Weg der Transaktion in die Blockchain* erörtert werden, dazu kommen, dass ein Block von mehreren Blöcken als Parent referenziert wird.

Merkle-Tree

Ein Merkle-Tree ist eine binäre Baumstruktur, bestehend aus den Hashwerten der Transaktionen des Blocks als Blätter.

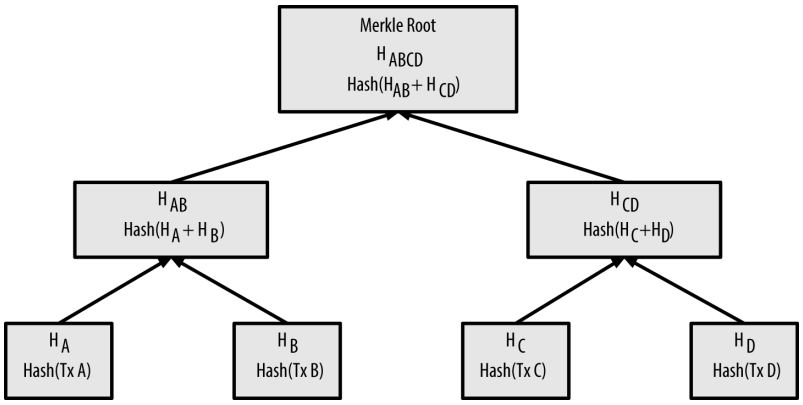

Abbildung 2.8.: Ein Merkle-Tree
Quelle: Antonopoulos (2014)

Auf jeder Ebene werden die Knoten paarweise gehasht, sodass man als Ergebnis die Wurzel als Identität aller Transaktionen im Block erhält. Aufgrund einer grundlegenden Eigenschaft von Hash-Funktionen, würde eine minimale Änderung an den Transaktionen die Wurzel, und somit den gesamten Block-Hash völlig verändern. Zu beachten ist noch, dass bei einer ungeraden Zahl von Transaktionen die letzte Transaktion in der Liste doppelt im Merkle-Tree aufgeführt wird, um das paarweise Hashing zu ermöglichen.

2.3.7 Der Weg der Transaktion in die Blockchain

Als Folge der dezentralen Struktur des Bitcoin-Netzwerkes ist ein Mechanismus nötig, der Transaktionen ohne ein zentrales Clearinghaus validiert und freigibt. An dieser Stelle kommt das *Mining* ins Spiel, über das zwar neue Bitcoin in Umlauf kommen, aber vor allem ein gemeinsamer Konsens im ganzen Netzwerk erreicht wird. Dieses Kapitel

beschäftigt sich mit den nötigen Schritten für einzelne Transaktionen bis hin zum neuen Block und welche Rolle Miner dabei spielen.

Miner und ihre Entlohnung

Miner sind spezielle Teilnehmer im Netzwerk, die darin propagierte Transaktionen aggregieren, validieren und miteinander darum konkurrieren, den nächsten Block zur Blockchain hinzufügen zu dürfen. Der Anreiz besteht darin, dass sie für ihren Aufwand, bei erfolgreicher Erweiterung der Blockchain, eine Entlohnung erhalten, die sich aus neuen Bitcoin und gesammelten Transaktionskosten ergibt.
Eine Tatsache, mit der Miner leben müssen ist, dass ihre Entlohnung mit der Zeit sinkt. Alle 210 000 Blöcke wird die Anzahl neuer Bitcoin halbiert, sodass die Gesamtzahl aller, sich im Umlauf befindenden Bitcoin, gegen 21 Millionen konvergiert. Aus diesem Grund wird Bitcoin als *Deflationäre Währung* bezeichnet, die mit der Zeit nicht an Wert verliert, sondern wertvoller wird.

Sammeln von Transaktionen

Genau wie alle anderen Teilnehmer sammeln Miner die im Netzwerk propagierten Transaktionen und überprüfen diese. Sollten jene bestimmte Kriterien, wie z.B. eine korrekte Syntax und Korrektheit, nicht erfüllen, werden sie nicht weitergeleitet und verworfen. Bei Korrektheit sammeln die Teilnehmer validierte Transaktionen in einem Pool und entfernen sie erst dann, wenn ein neuer und valider Block eine der gesammelten Transaktionen beinhaltet. Während alle anderen Teilnehmer bei Ankunft eines neuen Blocks lediglich die Transaktionen abgleichen und aus dem Pool entfernen, sammeln Miner übrig gebliebene Transaktionen und starten den Versuch, einen eigenen Block herzustellen.

Konstruktion des Blocks

Einen neuen Block zu erstellen ist für Miner aus zwei Gründen attraktiv:

- Sie erhalten alle Transaktionskosten aus den gesammelten Transaktionen
- Sie dürfen die sogenannte *Coinbase Transaction* selbst an den Anfang der gelisteten Transaktionen einfügen

Das besondere an dieser Transaktion ist, dass sie keine UTXO konsumiert, sondern dem Empfänger neu-generierte Bitcoin ausstellt. Sie augmentiert also nicht das UTXO-Set, sondern erweitert es durch neue Outputs. Nachdem Miner eine Liste von Transaktionen

inklusive Coinbase angelegt haben, konstruieren sie den Block Header bestehend aus Version, Hash des Vorgängerblocks, Wurzel aus dem Merkle-Tree, Zeitstempel, sowie Target und Nonce, die beim Mining eine Rolle spielen, erstellen.

Mining

Das Bitcoin-Protokoll ist so konzipiert, dass neue Blöcke in Intervallen von ca. 10 Minuten hinzugefügt werden. Um dem Anstieg an Rechenleistung vorzubeugen (man denke an Moore's Law), besitzt das Protokoll die Variable 'Target' als einen eingebauten Schwierigkeitsregler, der je nach Bedarf angepasst werden kann. Beim Target handelt es sich lediglich um eine Zahl, für die ein darunter liegender Wert durch wiederholtes Hashing des Block Headers, der sich ebenfalls serialisieren lässt, von den Minern gefunden werden muss. Falls der Miner keinen geeigneten Hash findet, bedient er sich an der Variable 'Nonce' die als Zählvariable dient und erhöht diese wiederholt, bis es zum Erfolg kommt. Alle 2016 Blocks wird die gesamte verstrichene Zeit mit der angestrebten Zeit von $2016 * 10Min$ verglichen und abhängig davon das Target entweder einfacher oder schwerer gemacht. Konkret bedeutet dies, den Wert anzupassen, um die Anzahl möglicher darunterliegender Hashwerte entweder zu reduzieren oder vergrößern. Dieses Verfahren funktioniert, weil ein Hashalgorithmus schon bei einem minimal veränderten Input einen gänzlich anderen Output zurückgibt. Gleichzeitig macht es die Fertigung eines Blocks im Voraus unmöglich, da der Hashwert des Vorgängerblocks im neuen Block Header enthalten ist. Im wesentlichen besteht das Verfahren also aus dem Konstruieren eines Blocks inklusive Block Header, dessen Hashwert, welcher das Target erfüllt, als 'Proof-of-Work' bezeichnet wird.

Während das 'Finden' eines geeigneten Blocks aufwendig ist, müssen andere Teilnehmer den SHA256-Algorithmus nur einmal auf den Block Header anwenden, um den gesamten Block zu validieren. Abgesehen vom Hash muss der Block allerdings noch andere Kriterien erfüllen, wie z.B. richtige Zeitstempel oder eine valide Liste von Transaktionen, inklusive valider Coinbase. Bei Gutschreibung von zu viel neuen Bitcoin wird der gesamte Block invalide, was betrügerisches Verhalten für Miner unattraktiv macht. Der Block wird nur im Netzwerk propagiert, wenn er die Kriterien erfüllt.

Nachdem ein Teilnehmer des Netzwerkes einen neuen Block validiert hat, muss er diesen zu seiner persönlichen Blockchain hinzufügen. Für den neuen Block gibt es drei Szenarios, wie die folgende Abbildung 2.9. zeigt:
Das erste Szenario ist das Erweitern der Mainchain, also der längsten Kette, und ist dann

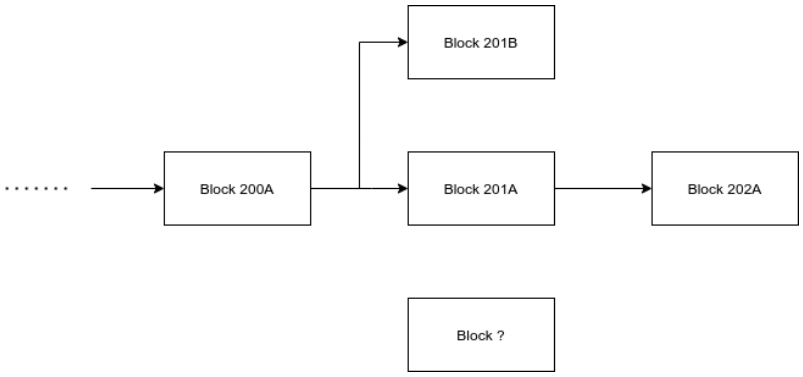

Abbildung 2.9.: Anordnung der Blöcke in einer Kette
Quelle: Eigene Darstellung

der Fall, wenn der neue Block (hier 202A) den Kopf dieser als Eltern-Block referenziert. Die zweite Möglichkeit besteht darin, dass der neue Block an einem inneren Block ansetzt und es dadurch zu einer Verzweigung, also einer neuen Sidechain kommt (hier 201B). Auch wenn dies im ersten Moment unnötig wirkt, werden die Sidechains angesammelt und bei jedem neuen Block mit der Mainchain verglichen, sodass sie, falls mehr Blöcke in diesen enthalten sind, zur neuen Mainchain werden. Die dritte und letzte Möglichkeit für neue Blöcke (abgesehen von invaliden Blöcken, die verworfen werden) sind sogenannte *Orphans*, also verwaiste Blöcke. Diese referenzieren einen Eltern-Block, der unbekannt ist, weshalb sie bis zur Veröffentlichung dessen in einem Pool gehalten werden. Dies kann dann der Fall sein, wenn Eltern- und Kind-Block in kurzer Zeit hintereinander gefunden werden und als Folge dessen bei manchen Teilnehmern in umgekehrter Reihenfolge ankommen. Auch wenn es in Folge des Minings zu Verzweigungen kommt, finden die Teilnehmer im Netzwerk immer einen gemeinsamen Konsens, indem sie der längsten Validen Kette, die am meisten Proof-of-Work 'angesammelt' hat, folgen. Miner konzentrieren sich dann darauf, für diese Kette neue Blöcke zu finden.

Wenn zwei Miner, die man auch als Kontrahenten in einem Wettkampf sehen kann, in relativ kurzer Zeit jeweils einen neuen Block finden, kann es zu einem sogenannten *Fork* kommen. Beide propagieren ihren gefundenen Block als neuen Kopf der längsten Chain und nehmen den Block des Gegners in einer Sidechain auf. Abhängig davon, welchen Block sie zuerst erhalten, folgen die anderen Teilnehmer dem Sender des ersten Blocks und geben ihm ihr Stimmrecht. Selbst bei einer gleichmäßigen Verteilung des Stimmrechts findet sich nach weiteren Generationen schließlich die Wahre Mainchain, da das gleichzeitigen Blöcken relativ selten ist. Eine zweite Möglichkeit für Forks sind sogenannte *Hard Forks*, bei denen man eine Sidechain in dem Wissen weiterführt, dass sie nie zur Mainchain wird. Dies kann beispielsweise der Fall sein, wenn eine Gruppe von Teilnehmern nicht mit der Funktionsweise des Protokolls zufrieden sind und eine erweiterte

Blockgröße fordern, um mehr Transaktionen zuzulassen. Man schreibt also ein neues Protokoll, welches eine Sidechain fortführt und von den daran interessierten Nutzern genutzt wird. Dies war der Hintergrund zur Entstehung von *Bitcoin Cash*.

2.3.8 Angriff auf das Netzwerk

Um einen bösartigen Angriff auf das dezentrale System ausüben zu können, muss man 51% der dem Netzwerk zur Verfügung stehenden Rechenleistung besitzen. Ist dieser Umstand gegeben, haben Angreifer folgende Optionen:

- Es kann ein 'Double Spending' ausführt werden
- Einzelnen Teilnehmern kann die Anteilnahme am Netzwerk untersagt werden

Unter normalen Umständen ist Double Spending nicht möglich, doch Angreifer könnten Bitcoin auf eine Handelsbörse überweisen und dort in einer gewünschte Währung auszahlen lassen. Anschließend setzen sie mit einer Sidechain dort an, wo sich das Guthaben noch beim Angreifer befindet, also die UTXO ungenutzt sind, und erweitern diese zur neuen Mainchain. Die zweite Möglichkeit des Angriffs besteht darin, dass einzelne Teilnehmer von der Anteilnahme ausgeschlossen werden, indem man beispielsweise ihre Transaktionen nicht in neue Blöcke aufnimmt. Das Gleiche gilt für bestimmte Miner, deren Blöcke man gezielt ablehnen kann.

Aufgrund der immensen Rechenleistung, die dafür nötig wäre, wird ein Angriff in tieferen Schichten teurer und wäre, abgesehen von durch Konzerne oder Staaten finanzierten Attacken, unwahrscheinlich.

3 Blockchain 2.0

Durch den Zusammenschluss verschiedenster Vorgängertechnologien gelang es Nakamoto, einen dezentralen Mechanismus zur Verarbeitung von Transaktionen vorzustellen, der mit den bereits bestehenden Institutionen in Punkten wie Sicherheit und Geschwindigkeit konkurrieren kann. Vor allem die Möglichkeit, auf eine dezentrale Art und Weise zu einem gemeinsamen Konsens kommen zu können, macht es zu einer disruptiven Technologie. Als Programmierer jedoch Konzepte, die über das Versenden von Guthaben hinausgehen, entwickeln wollten, wurde dies durch die Einschränkungen des Bitcoin-Protokolls, wie die Block-Dauer von 10 Minuten oder unzureichende Block-Struktur, erschwert. Man kann deshalb von Bitcoin als spezialisierte Blockchain reden, die ihre wenigen Funktionalitäten zufriedenstellend bereitstellt.

Im Jahr 2013 veröffentlichte Vitalik Buterin das Whitepaper Buterin (2013b), in dem er Konzepte für eine neue Blockchain namens Ethereum vorstellte. Dabei handelt es sich um den Entwurf für eine Art Welt-Computer, der sich von Bitcoin vor Allem in den folgenden Punkten unterscheidet:

1. Es können beliebige Daten auf der Blockchain gespeichert werden
2. Es kann Code auf der Blockchain hinterlegt und dort ausgeführt werden

Im Gegensatz zum Bitcoin-Protokoll, welches zwischen den Blöcken nur eine Veränderung des UTXO-Sets aufweist, stellt Ethereum einen globalen State dar, in dem neben den Aufzeichnungen zu den Guthaben der Teilnehmer auch andere beliebige Daten abgespeichert werden können. Damit geht Ethereum über den Einsatz als Geldsystem hinaus. Über spezielle Adressen erlaubt das Ethereum-Protokoll das hinterlegen von kompilierten Code direkt auf der Blockchain, welcher über Transaktionen getriggert werden und den State modifizieren kann.

Ethereum übernimmt viele Konzepte, die bereits aus dem Kapitel *Blockchain 1.0* bekannt sind (Keys, Wallets, Mining, etc.). Dieses Kapitel befasst sich mit Neuerungen, die das Ausführen von Smart-Contracts ermöglichen bzw. durch diese ermöglicht werden. Außerdem werden mögliche Use-Cases für Smart-Contracts beschrieben.

3.1 Neuerungen unter Ethereum

Im Gegensatz zum Bitcoin-Protokoll, in dem das UTXO-Modell genutzt wird, kommt beim Ethereum-Protokoll ein sogenanntes *Account Based Model* zum Einsatz. Jede Adresse auf der Blockchain hat ein Guthaben, auf das Ether, die native Währung auf Ethereum, hinzugefügt bzw. abgehoben werden kann. Eine passende Analogie für die beiden Kontenmodelle wäre das UTXO-Modell als physikalische Geldbörse, aus der für eine Transaktion einzelne Münzen (UTXO) herausgesucht und konsumiert werden müssen, sowie das Girokonto, dessen Guthaben beliebig augmentiert werden kann. Dies hat zur Folge, dass andere Mechanismen zum Lösen des Double-Spending-Problems geben muss.

3.1.1 Transaktionen

Neben dem Transfer von Guthaben, so wie es bereits von Bitcoin bekannt ist, dienen Transaktionen auf Ethereum dem Zweck, die Ausführung von, auf der Blockchain hinterlegten, Smart-Contracts auszulösen. Um diese Funktion erfüllen zu können, unterscheidet sich eine Transaktion auf Ethereum zu deren im Bitcoin-Protokoll:

```
1  Transaction {
2    nonce;
3    recipient;
4    value;
5    data;
6    v;
7    r;
8    s;
9    gas_price;
10   gas_limit;
11 }
```

Nonce ist ein Zähler, der bei Transaktionen auf Ethereum eine andere Bedeutung hat, als beim Mining. In einer Transaktion gibt er an, wie viele bestätigte Transaktionen, inklusive sich selbst, bereits von jener Adresse gesendet wurden. Dies führt dazu, dass das Netzwerk eine Warteschlange, aus den von einer Adresse gesendeten Transaktionen, bilden kann. Da im Account-Based-Modell keine UTXO konsumiert werden, kann das Netzwerk anhand des Zählers eine Priorisierung der unbestätigten Transaktionen festlegen und, sollte das gesamte Guthaben nicht ausreichen, Transaktionen mit niedrigerer Priorität ablehnen. Der eigentliche Payload einer Transaktion besteht aus *Value* und *Data*, wobei keines dieser Felder zwingend gefüllt sein muss. Als Value bezeichnet man die Ether, welche bei

einer Transaktion gesendet werden können und je nach Art der Empfängeradresse anders verarbeitet werden. Findet die Transaktion zwischen zwei Teilnehmern des Netzwerkes statt, so werden die Ether der Empfängeradresse zugeschrieben. Handelt es sich dagegen um einen Smart-Contract, so wird dessen Guthaben (Ein Smart-Contract hat eine Adresse und kann dementsprechend Ether halten) erhöht und eine, als *payable* gekennzeichnete, Fallback-Funktion ausgeführt. Bei Fehlen einer solchen Funktion wird ein Fehler geworfen und andernfalls der State des Smart-Contracts angepasst. Die Fallback-Funktion nur dann ausgeführt, wenn das Feld Data des Payloads leer ist. In diesem selektiert man die gewünschten Funktionen und mit welchen Parametern diese ausgeführt werden sollen. Die Felder *v, r ,s* sind für die Authentifizierung nötig, werden hier aber nicht weiter erörtert. Im Endeffekt werden Transaktionen wie bekannt mit dem privaten Schlüssel signiert und andere Teilnehmer des Netzwerkes validieren diese mit dem öffentlichen Schlüssel.

Neues Konzept: Gas

In seinem Paper Turing (1936) stellte Alan Turing einen Entwurf für eine Maschine vor, die aus einem nahezu unendlich langem Streifen aus Papier und einem Kopf besteht. Der Streifen ist in Bereiche unterteilt, die jeweils ein Zeichen, wie z.B. eine Zahl, enthalten und zwischen denen sich der Kopf bewegen kann. Dieser ist in der Lage, das Zeichen des Bereiches, 'auf dem er sich befindet', zu lesen, zu überschreiben, zu löschen oder auch auf dem Streifen zu verschieben. Die Maschine ermöglicht somit, in Abhängigkeit zu den gegebenen Instruktionen, das Berechnen sowie abspeichern beliebiger Sequenzen von Zeichen auf dem Streifen. Eine solche Maschine wird heutzutage als Turing-Maschine bezeichnet.

Im Ethereum-Whitepaper Buterin (2013b) wird Ethereum als Blockchain bezeichnet, die eine Programmiersprache verwendet, welche turing-vollständig ist. Michaelson beschreibt in Michaelson (2020) turing-vollständige Programmiersprachen als solche, die ihre Berechnungen in Form einer Turing-Maschine ausdrücken können. Dafür dürfen sie laut Michaelson weder in ihren genutzten Werten, noch in den darauf ausgeführten Berechnungen beschränkt sein. Konkret heißt das im Fall von Ethereum und der genutzten Programmiersprache, dass beliebig aufwendige Berechnungen, sowie geforderter Speicherbedarf ermöglicht werden müssen. Auf einem lokalen Computer kann eine endlos laufende Schleife leicht gestoppt werden, aber im Kontext eines dezentralen Systems würde jene Schleife die Stabilität des gesamten Netzwerkes bedrohen und würde sogar als Angriffsmethode für bösartige Akteure dienen können. Im Endeffekt nutzen die Teil-

nehmer des Netzwerkes Ressourcen aus dem selben Pool, sodass ein Mechanismus nötig ist, um das Verschwenden dieser begrenzten Ressourcen unattraktiv zu machen.

Die Anweisungen, welche in einem Smart-Contract hinterlegt sind, werden nach Triggern durch eine Transaktion von der *Ethereum Virtual Machine* interpretiert und ausgeführt. Jede der Anweisungen verbraucht jedoch *Gas*, welches als neues Konzept zur Lösung der genannten Probleme aus Touring-Vollständigkeit eingeführt wurde. So bleibt die Programmiersprache zwar touring-vollständig, die Ressourcen, welche Teilnehmer in Anspruch nehmen können, werden jedoch begrenzt. Dies hat zur Folge, dass die Anzahl an Gas, sowie der Preis in Ether, welchen man bereit ist zu zahlen, in einer Transaktion angegeben werden muss. Miner suchen sich anschließend die Transaktionen heraus, welche für sie attraktiv sind, und fügen sie zu ihrem Block hinzu. Übrig gebliebenes Gas wird in Form von Ether wieder zurückerstattet. Zu einer Transaktion gehören demnach die, für den Versand bestimmten, Ether, sowie jene, welche für den Erwerb von Gas bereitgestellt werden müssen.

3.1.2 Ethereum Virtual Machine

Genau wie im Bitcoin-Protokoll sammeln Miner Transaktionen, validieren diese und versuchen, per Proof-Of-Work einen geeigneten Block zu finden und im Netzwerk zu propagieren. Im Gegensatz zu den Transaktionen im Bitcoin-Protokoll lösen die Transaktionen unter Ethereum allerdings die Ausführung von, auf der Blockchain hinterlegten, Smart-Contracts aus. Für die Ausführung derer ist die Ethereum Virtual Machine (kurz EVM) zuständig. Sie ist eine Virtuelle Maschine, die in Wood (2014) als stack-basierte Architektur beschrieben wird. Die folgende Abbildung illustriert eine vereinfachte Darstellung ihrer Funktionsweise:

Die EVM erhält als externen Input die Daten für Transaktionen, die sie ausführen muss und wandelt die darin enthaltenen Funktionsaufrufe sowie ihre Parameter in ausführbare Einzeloperationen in Form von Maschinencode um. Als zusätzliche Inputs für die Verarbeitung von Transaktionen dienen der Smart-Contract-Code, als auch Umgebungsdaten, wie z.B. die verfügbare Menge an Gas oder das Guthaben der involvierten Adressen. Die EVM führt die Anweisungen nach dem LIFO-Prinzip im Kontext der Umgebungsdaten aus und verbraucht für jede Anweisung eine gewisse Menge Gas, die von der Art jener Operation abhängt. Wenn während der Ausführung keine Fehler auftreten, dann wird der State persistiert und mit der nächsten Transaktion fortgefahren. Wenn allerdings an einer beliebigen Stelle des Stacks ein Fehler auftritt, sei es beispielsweise durch unzureichendes

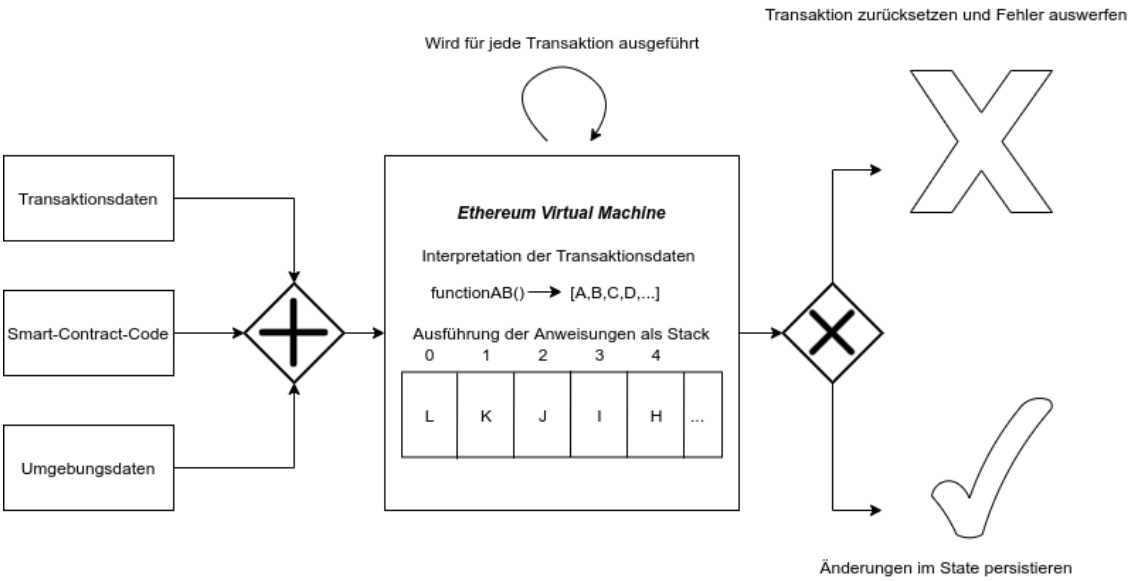

Abbildung 3.1.: Vereinfachung der Ethereum Virtual Machine

Gas, wird die gesamte Transaktion zurückgesetzt, da Transaktionen atomar sind und nur in ihrer Gesamtheit ausgeführt werden können. Erst wenn ein valider neuer State erreicht ist, gilt der Block selbst als valide und kann mittels Proof-Of-Work gemined werden. Im Anhang unter A.1 befindet sich eine umfassende Abbildung aus dem Buch Antonopoulos (2018).

3.1.3 Die Programmiersprache Solidity

Der Programmcode, welcher von der EVM interpretiert wird, ist in der Programmiersprache *Solidity*, welche an andere objektorientierten Programmiersprachen wie Java oder Javascript erinnert, geschrieben. Sie bietet bekannte Funktionalitäten wie Kontrollstrukturen, Vererbung, etc. an, wobei man aufgrund der Verwendung von Gas auf die Wirtschaftlichkeit des Codes achten. Anhand des folgenden Codes sollen kurz die Syntax von Solidity dargestellt werden:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.3;
3
4  contract AdvertContract {
5    address owner;
6    struct Ad {
7      uint counter;
8      uint funds;
9    }
10   mapping(uint => Ad) public ads;
```

```
11
12      constructor() {
13        owner = msg.sender;
14      }
15
16      receive() external payable {}
17      function getBalance() public view returns(uint){
18        return address(this).balance;
19      }
20  }
```

Zunächst werden die Lizenz, als auch die Version des gewünschten Compilers definiert [Zeile 1-2]. Der Befehl *Contract* lässt sich mit einer Klasse aus anderen Programmiersprachen vergleichen. Es können einfache Felder[5], eigene Datenstrukturen [6-9], als auch sogenannte *mappings* definiert werden [10]. Mappings lassen sich mit Hash-Tabellen vergleichen und über die Notation (uint => Ad) gibt man an, dass über einen uint, welcher für unsigned Integer steht, als Schlüssel auf Objekte des Typs Ad zugegriffen werden kann. Der Contract verfügt über einen Konstruktor, der nur einmal während des Deployments aufgerufen wird. In diesem kann beispielsweise diejenige Adresse festgelegt werden, die als Eigentümer des Contracts gelten soll. Während des Deployments wird eine spezielle Transaktion, welche den kompilierten Code enthält, an die Adresse 0x00... gesendet und von den Minern als Aufforderung für das Hinterlegen des Codes auf der Blockchain interpretieren. Die globale Variable *msg* ermöglicht den Zugriff innerhalb des Contracts auf Informationen der Transaktion, welche diesen triggert [12-14]. *Receive* stellt eine Fallback-Funktion dar, welche das Verhalten des Contracts für den Fall definiert, dass eine Transaktion mit leerem Feld 'data' diesen aufruft, was einer schlichten Überweisung gleicht [16]. Der Decorator *payable* ist dann nötig, wenn eine, an diese Funktion gerichtete, Transaktion zusätzliche Ether mitschickt, denn anderenfalls wird ein Fehler ausgeworfen. *External* macht, ähnlich wie der Decorator public, die Funktion nach außen hin sichtbar, ist jedoch bei der Verwendung von Gas effizienter. Über den Befehl *function* lassen sich eigene Funktionen definieren für die man gewünschte Parameter, Zugriffsmodifikator angeben muss. Für den Fall, dass die Funktion einen Wert zurückgibt, muss ein Rückgabewert und bei ausschließlich lesendem Zugriff der Befehl *view* angegeben werden[17-19].

3.1.4 Mögliche Use-Cases

DAO

Buterin (2013b) bezeichnet *Decentralized Autonomous Organizations* als dezentrale Organisationen, in denen Parteien mit einer Mehrheit von 67% des Stimmrechts über die Zukunft der Organisation, welche in Smart-Contracts definiert ist, entscheiden dürfen. Darin ist festgelegt, wer zur Organisation gehört und ob bestimmte Mitglieder von ihr bezahlt werden. Eine 67% Mehrheit könnte neue Mitglieder aufnehmen, entlassen oder deren Bezahlung anpassen.

Tokens

Als Token bezeichnet Antonopoulos (2018) die Abstraktion einer bestimmten Sache auf der Blockchain, die man besitzen kann. Dies können wieder Währungen zum Bezahlen innerhalb des Systems sein, als auch physikalische Besitztümer wie Gold oder Immobilien. Token, welche einzigartige physikalische, als auch digitale Gegenstände repräsentieren, werden *Non-fungible Token* genannt und der Vorteil dieser Token ist, dass der Besitz nicht durch eine zentrale Partei wie z.B. einen Staat, sondern durch den Besitz des dazugehörigen Keys definiert wird.

DApps

Klassische (Web-)Applikation bestehen meist aus den Front-, Backend und Datenbank. Das Vorhaben hinter *Decentralized Applications* ist eine Dezentralisierung von Anwendungen, indem man bestimmte Funktionalitäten über Smart-Contracts oder andere dezentrale Mechanismen bereitstellt. Dies hat den Vorteil, dass man sowohl Transparenz schafft (Die Blockchain ist öffentlich und kann von jedem Akteur eingelesen werden), als auch die Abhängigkeit von Parteien, welche für die Infrastruktur der Anwendung verantwortlich sind, minimiert.

4 Blockchain im Online Advertising

Die bisherigen Kapitel dienen dem Zweck, Lesern ein Verständnis über technische, als auch funktionelle Aspekte der Blockchain-Technologie zu vermitteln. Allerdings ist Informationstechnologie kein Selbstzweck, sondern wird zur Lösung von konkreten Problemen verwendet. Im folgenden Kapitel soll diese, im Kontext des *Online Advertisings*, in einen wirtschaftlichen Prozess sinnvoll eingebaut werden. Dafür wird zunächst das Thema Online Advertising näher beschrieben und mögliche Verbesserungen mittels Blockchain-Technologie erörtert. Ausgehend davon wird die Architektur und ihre Komponenten für eine Webanwendung, in die ein Smart-Contract eingebaut ist, vorgestellt. Abschließend wird die Eignung von Blockchain-Technologie auf ihrem derzeitigen Stand für das Online Advertising erörtert und derzeitige Probleme, sowie mögliche Lösungen beschrieben.

4.1 Online Advertising

Laut Johnson (2021) ist das Internet heutzutage fester Bestandteil des Lebens der 4,66 Milliarden Menschen auf der Erde, die es regelmäßig verwenden. Ein solch hoher Traffic macht das Schalten von Werbung attraktiv, sodass im Jahr 2021 laut Statista (2021) ein Umsatz von 139,8 Milliarden US-Dollar in den USA erzielt werden konnte. Die verschieden Arten von Werbung werden unter dem Begriff *Online Advertising* gesammelt. Im Paper Bundeskartellamt (2018) des Bundeskartellamts wird Online Advertising als jegliche Art Werbung, die über das Internet auf mobilen, sowie Desktopanwendungen vermittelt wird, bezeichnet. Darin werden auch die relevante Unterkategorien beschrieben:

- Search Advertising: Werbeanzeigen werden auf den Oberflächen von Suchmaschinen entweder als Anzeigen seitlich der Suchergebnisse oder als Bestandteil dieser angezeigt
- Mobile Advertising: Auf Mobilgeräte angepasste Anzeigen, die auch Bestandteile von Apps sein können
- Social media Advertising: Nutzer mit einer hohen Reichweite bauen Werbung in ihre Inhalte ein. Diese Nutzer nennt man auch *Influencer*.
- Display Advertising: Inhaber von Internetseiten bieten verfügbaren Platz als Werbeflächen an

Mit der ersten online-geschalteten Werbeanzeige im Jahr 1994 ist das Display Advertising die älteste Form des Online Advertisings und soll im Folgenden näher thematisiert werden Bundeskartellamt (2018).

4.1.1 Display Advertising Heute

Die einfachste Form des Display Advertisings wäre die Übereinkunft zwischen Anbieter, welcher Werbung auf der Internetseite schaltet und werbendem Unternehmen, welches pro Schaltung bezahlt. In der Praxis würde dies allerdings einen hohen Aufwand für beide Parteien bedeuten. Unternehmen müssten einen Vertrag darüber aufsetzen, welche Werbung wie oft geschaltet werden sollte und Anbieter der Werbefläche müssten sich selbst um das Darstellen der Anzeigen, als auch Tracking diverser Metriken, wie z.B. Impressionen oder Klicks, kümmern. Stattdessen nimmt man die Dienste eines sogenannten *Ad-Brokers* in Anspruch, sodass die folgende Konstellation entsteht:

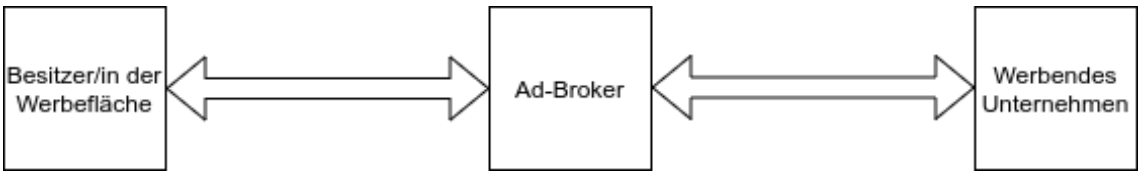

Abbildung 4.1.: Involvierte Parteien in gängigem Display Advertising

Der Ad-Broker dient als Intermediär zwischen Angebot und Nachfrage, indem dieser einen Marktplatz für die werbenden Parteien zur Verfügung stellt. Ein Beispiel hierfür ist Googles *AdSense* welches Anbieter mit einem Code-Snippets versorgt, die sie lediglich im Quellcode ihrer Internetseite einfügen müssen. Über diese werden Anzeigen direkt auf die Seite geladen, ohne dass die Anbieter sich selbst darum kümmern müssen. Zur Unternehmensseite hin fungiert AdSense als Marktplatz, auf dem Unternehmen Gebote für die, mittels Code-Snippet, bereitgestellten Anzeigeflächen abgeben. Zusätzlich dazu können Unternehmen diverse Metriken für erworbene Werbeflächen abrufen, die von AdSense getrackt werden.

4.1.2 Mögliche Verbesserungen mittels Blockchain-Technologie

Auch wenn die Einbeziehung eines Ad-Brokers den vermittelten Parteien Aufwand erspart, entstehen dadurch nicht nur Vorteile. Für Anbieter von Werbeflächen entsteht der Nachteil, dass nicht das gesamte Geld des Unternehmens bei ihnen ankommt, weil der Ad-Broker gleichzeitig den Zahlungskanal darstellt. Stattdessen erhalten diese laut Google (2021) 68% des Umsatzes, wodurch in diesem Fall fast ein Drittel an den Ad-Broker Google geht. Auf Unternehmensseite entsteht der Nachteil fehlender Transparenz, denn Zugriff auf die getrackten Metriken erhält man nur über den Ad-Broker.
Blockchain-Technologie könnte diese Probleme lösen, indem die Rahmenbedingungen für Interaktionen zwischen Anbietern und Unternehmen mittels Smart-Contracts geregelt wird. Da Ethereum das hinterlegen beliebiger Daten ermöglicht, könnte in den Smart-

Contracts aufgezeichnet werden, wie oft eine Anzeige geschaltet wurde und welche Kosten dafür aufkommen. Aufgrund der Transparenz einer öffentlichen Blockchain können Unternehmen jederzeit auf darauf hinterlegte Metriken abrufen. Gleichzeitig kann die Blockchain in ihrer ursprünglichen Funktion als direkter Zahlungskanal zwischen Anbietern und Unternehmen genutzt werden. Als Folge dessen würde der Ad-Broker als Intermediär wegfallen und Vorteile für beide übrigen Parteien würden entstehen.

4.2 Programmierung eines PoC

Die Chancen, welche Blockchain-Technologie für das Display Advertising bieten könnte, sollen im Folgenden am Beispiel eines *Proof-of-Concept* veranschaulicht werden. Dafür wird eine Webanwendung entwickelt, die beliebigen Inhalt abbildet und an den Seiten Platz zum Schalten von Anzeigen hat. Zusätzlich dazu sollen die soeben genannten Verbesserungen eingebaut werden, indem Unternehmen direkt auf der Seite eine Anzeige hochladen und eine Art Guthaben erwerben können, die beim Schalten der Anzeige aufgebraucht wird. Die Unternehmen sollen in der Lage sein, die Anzahl der Schaltungen, als auch das übrige Guthaben abrufen und bei Bedarf erhöhen zu können. Die Webanwendung soll den folgenden Aufbau haben:

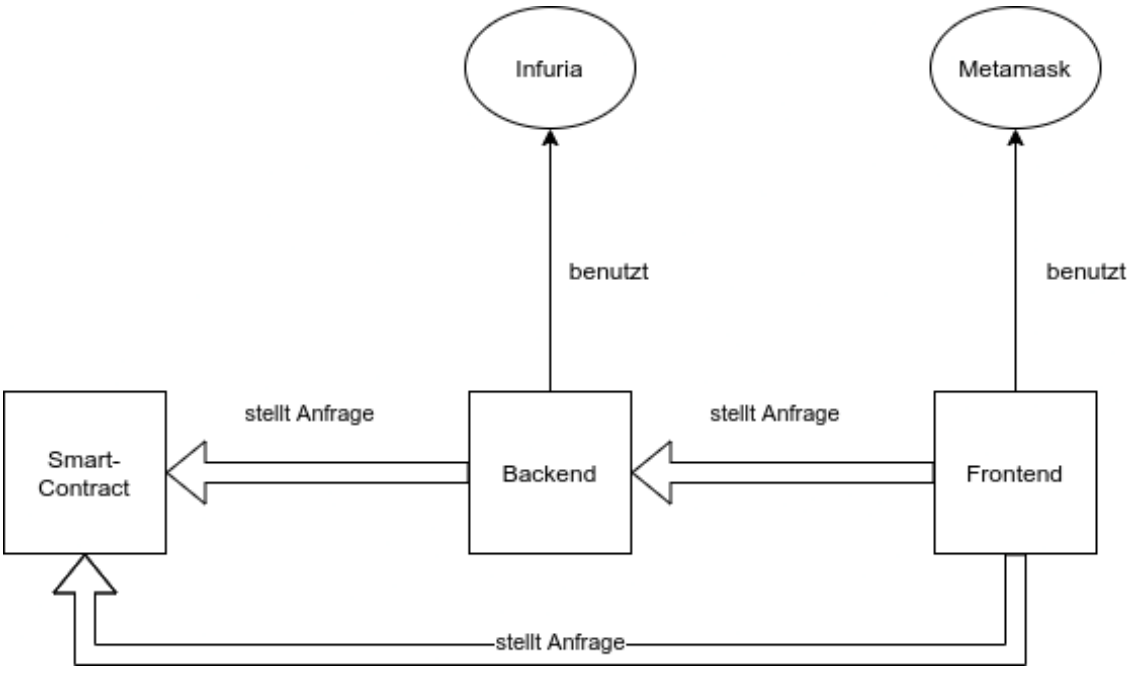

Abbildung 4.2.: Aufbau der Webanwendung

Die Anwendung besteht aus einem Front- und Backend, sowie einem Smart-Contract auf dem Ropsten-Testnetz, welches zum Testen von Smart-Contracts unter Ethereum dient. Sowohl Front-, als auch Backend können nicht direkt mit der Blockchain kommunizieren, sondern tun dies über sogenannte *Provider*. In den folgenden Unterkapiteln sollen die

einzelnen Komponenten näher erläutert werden.

4.2.1 Frontend

Das Frontend bildet den Teil der Anwendung, welcher im Browser von Nutzern läuft. Es kümmert sich um die Visualisierung und Interaktionen mit den Nutzern der Seite. Für das Frontend wird *Angular*, welches von Google in Stand gehalten wird, genutzt. Über einen einzigen Befehl *ng new* auf der Kommandozeile, lässt sich eine lauffähige Webanwendung generieren, die anschließend nach Belieben überarbeitet werden kann. Außerdem wird statt Javascript die Programmiersprache *Typescript* genutzt, welche den Vorteil der Typsicherheit, wie sie beispielsweise aus Java bekannt ist, mitbringt. Ein letzter Vorteil unter vielen ist die Nutzung der Bibliothek *RXJS*, welche eine Alternative zu bekannten Technologien wie Ajax zur Behandlung von asynchronen Funktionen bietet. Zur Kommunikation mit der Blockchain wird die Bibliothek *Ethers.js* genutzt. Sie erlaubt es Entwicklern, Anfragen an eine Blockchain-Node zu schicken, von der Transaktionen anschließend signiert und gesendet werden. Das Frontend lässt sich grob in die folgenden drei Elemente unterteilen:

- Die Hauptkomponente, bestehend aus HTML-, CSS- und Typescriptdatei
- Der ApiService, welcher sich um die Kommunikation mit dem Backend kümmert
- Der BlockchainService, welcher sich um die Kommunikation mit dem Smart-Contract kümmert

Die Hauptkomponente existiert bereits nach der initialen Generierung und dient als Gerüst für die Anwendung. In Angular können Komponenten nach dem Bausteinprinzip beliebig angeordnet werden, sodass jede Komponente für einen anderen Teil der Anwendung zuständig sein kann. Auch eine hierarchische Anordnung ist möglich, sodass eine Komponente wieder andere enthält. Sogenannte *Services* können durch Deklaration im Konstruktor von Komponenten genutzt werden, dies nennt man *Injecting*. Services sind keine eigenständigen Komponenten, sondern stellen lediglich Funktionalitäten bereit, wie z.B. die Kommunikation mit dem Backend. Einige wichtige Funktionen der Datei *app.component.ts* sollen im Folgenden erläutert werden.

Konstruktor und Laden von Bildern

```
1 public constructor(private sanitizer: DomSanitizer,
2 private api: ApiService,
3 private http: HttpClient,
4 private formBuilder: FormBuilder,
```

```
5  public blockService: BlockchainService) {
6    this.getImageFromApi();
7    this.getImageFromApi();
8  }
9
10 public getImageFromApi(): void {
11   this.api.getImage().subscribe((baseImage: any) => {
12     const blob = new Blob([baseImage]);
13     const unsafeImg = URL.createObjectURL(blob);
14
15     this.upperRight === undefined ?
16     this.upperRight = this.sanitizer
17             .bypassSecurityTrustUrl(unsafeImg) :
18     this.upperLeft = this.sanitizer
19     .bypassSecurityTrustUrl(unsafeImg);
20   });
21 }
```

Durch die Deklarationen von Klassen als Parameter des Konstruktors injiziert man diese, sodass Objekte der Klassen im Rumpf und allen anderen Methoden verwendet werden können [siehe Zeile 1-5]. So wird in der Methode *getImageFromApi()* die am Objekt der Klasse ApiService die Methode *getImage()* aufgerufen, die ein *Observable* zurückgibt. Observables kann man sich als Objekte von asynchronen Funktionen vorstellen, über die man mithilfe der Methode *subscribe()* festlegen kann, wie vorgegangen werden soll, sobald die asynchrone Funktion abgeschlossen ist [11]. Anschließend wird das rohe Bild aus der Antwort verarbeitet [12-13] und über einen ternären Operator wird entschieden, an welcher Stelle das geladene Bild eingefügt werden soll [15-17].

Anpassen des Guthabens

```
1   public augmentAds(): void {
2     const id = parseInt(this.idForFunds, 10);
3     const funds = parseInt(this.fundsToAdd, 10);
4
5     const payment = ethers.utils.parseEther(this.fundsToAdd);
6     const overrides = {
7       value: payment
8     };
9     this.blockService.augmentAds(id, funds, overrides).then(res => {
10      console.log('Funds Added!');
```

```
11      });
12    }
```

Die Methode *augmentAds()* ist dafür da, das bestehende Guthaben für eine Anzeige zu erhöhen. Dafür geben Nutzer über Inputfelder auf der Seite die Id ihrer Anzeige und die Menge von Guthaben an, die sie aufstocken möchten. In Angular können Inputfelder direkt mit öffentlichen Variablen der Komponente verknüpft werden, sodass *idForFunds* und *fundsToAdd* direkt abgerufen werden können [2-3]. Die Variable *payment*, welche das Guthaben umgewandelt in Ether repräsentiert, wird initialisiert und als Wert des Felds *value* in der Variable *overrides* verwendet [5-8]. *augmendAds()* der Klasse Blockchain-Service ruft die gleichnamige Methode des Smart-Contracts auf, die jedoch mit nur zwei Parametern deklariert ist. Mithilfe eines Dritten, wird in diesem Fall das Feld value im Payload der eigentlichen Transaktion überschrieben, sodass eine Zahlung in Ether, die dem gewünschten Guthaben entspricht, entsteht [9-11].

Ausschnitt : Hochladen einer Anzeige

```
1   // Ausschnitt aus der Methode 'onFormSubmit()'
2
3   const formData = new FormData();
4   formData.append('uploadedImage',
5           this.fileUploadForm.get('uploadedImage').value);
6   formData.append('etherSend',
7           this.fileUploadForm.get('etherSend').value);
8
9   this.blockService.sendMoney(this.fileUploadForm
10  .get('etherSend').value).then((res) => {
11    this.http.post<any>('http://localhost:3000/upload', formData)
12    .subscribe(response => {
13      console.log(response);
14      // An dieser Stelle kann die Response verarbeitet werden;
15    }, error => {
16      console.log(error);
17    });
18  });
19  }
```

Alternativ zum direkten Verknüpfen mit einer Variable können Inputfelder Teil einer *FormGroup* sein. Diese bieten Funktionalitäten, welche für Formulare nützlich sind. So können beispielsweise Validatoren für Felder, die bestimmte Voraussetzungen erfüllen

müssen, gesetzt werden. Über die Methode *get().value* wird das Inputfeld *etherSend* ausgelesen und als Parameter für die Methode *sendMoney* des Services verwendet [9-10]. Diese überschreibt genau wie augmentAds() den Payload Transaktion. Nach dem Senden wird ein Formular, bestehend aus Anzeige und Wert der Transaktion [3-7], an das Backend gesendet, in dem der Payload verarbeitet wird. Sobald eine Antwort aus dem Backend kommt, kann diese verarbeitet werden, indem man beispielsweise die Id der hochgeladenen Anzeige darstellt oder eine Fehlermeldung bei Misslingen anzeigt [11-18].

4.2.2 Backend

Das Backend ist für das Verwalten der Anzeigen zuständig und reagiert auf Anfragen aus dem Frontend. Geschrieben ist es in Javascript, wodurch der Vorteil entsteht, dass hier ebenfalls Ethers.js für die Kommunikation mit der Blockchain genutzt werden kann. Allein Javascript ist jedoch für ein Backend nicht ausreichend, denn die Programmiersprache wurde zur Ausführung im Browser entwickelt. *Node.js* löst dieses Problem, indem es eine Laufzeitumgebung zur Ausführung von Javascript bereitstellt, durch die Javascript-Dateien auf einer Konsole und somit einem Server ausgeführt werden können. Zusätzlich dazu wird das Framework *Express.js* verwendet, um ein lauffähiges Backend, welches auf Anfragen des Frontends reagieren kann, bereitstellen zu können. Eingehende Dateien werden mithilfe der Bibliothek *multer* verarbeitet. Für die Kommunikation mit dem Frontend werden zwei Schnittstellen zur Verfügung gestellt:

- Eine Schnittstelle zum Hochladen von Anzeigen
- Eine Schnittstelle für das Laden von Anzeigen

Genau wie beim Frontend sollen im Folgenden die wichtigsten Funktionen im Code näher erläutert werden

Hochladen einer Anzeige - Erfolg

```
1  // wird vorher initialisiert: const app = express();
2
3  app.post('/upload', upload.single('uploadedImage'),
4  (req, res) => {
5    const tempPath = req.file.path;
6    const index = getIndex();
7    const originalFilename = req.file.originalname
8          .substring(0, req.file.originalname.indexOf('.'));
```

```
9     const filename = originalFilename + index + '.jpg';
10    const targetPath = path.join(__dirname, "./uploads/"
11                    + filename);
12
13    if (path.extname(req.file.originalname).
14            toLowerCase() === ".jpg") {
15      fs.rename(tempPath, targetPath, err => {
16        if (err) return handleError(err, res);
17        res
18        .status(200)
19        .contentType("text/plain")
20        .end("Your Id is:" + getIndexAsNumber(index).toString());
21      });
```

Das Objekt *app* ist eine Instanz von Express, für die HTTP-Anfragen unter beliebigen Sub-Adressen definiert werden können. Im Fall des Hochladens wird eine POST-Anfrage unter der Sub-Adresse */upload* definiert und mithilfe von *upload*, welches eine Instanz von Multer ist, kann auf die Bilddatei des Uploads, die sich unter *uploadedImage* befindet, zugegriffen werden [3]. Anschließend wird der aktuelle Pfad der Datei in der Variable *tempPath* gespeichert und ein neuer Pfad *targetPath*, welcher die Id der Anzeige enthält, berechnet [5-11]. Wenn es sich um eine .png-Datei handelt, wird der aktuelle Pfad durch den berechneten ersetzt und eine Erfolgsmeldung, welche die berechnete Id der Anzeige enthält, als Antwort an das Frontend zurückgegeben.

```
1       const newAd = {
2         id: getIndexAsNumber(index),
3         // Funds werden von Ether in Gwei umgewandelt
4         funds: parseFloat(req.body['etherSend']) * (10 ** 9)
5       };
6       blockchain.newAd(newAd).then(res => console.log(res))
7       .catch(err => console.log(err));
8
9     } else {
10      fs.unlink(tempPath, err => {
11        if (err) return handleError(err, res);
12
13        res
14        .status(403)
15        .contentType("text/plain")
16        .end("Only .jpeg files are allowed!");
17      });
```

```
18      }
19    })
```

Nach dem erfolgreichen Speichern der Anzeige wird ein neuer Eintrag auf der Blockchain unter der neuen Id gesetzt. Dafür wird zunächst ein Objekt namens *newAd* initialisiert, welches die Id als Ganzzahl und das gewünschte Guthaben in *Gwei* enthält. Eine Milliarde Gwei, was abgekürzt Gigawei bedeutet, entsprechen einem Ether und eine Milliarde Wei, die kleinste Einheit von Ether, entsprechen wiederum einem Gwei [1-5]. Bei dem Objekt *blockchain* handelt es sich um eine Sammlung von Funktionen, die aus einer anderen Datei exportiert wurden. Diese Datei ist vergleichbar mit den Services aus dem Frontend. An diesem Objekt wird die Funktion *newAd* aufgerufen und das initialisierte Objekt als Parameter mitgegeben [6-7]. Falls es sich bei der Datei nicht um eine .png-Datei handelt, wird diese gelöscht [10] und eine Fehlermeldung ausgeworfen [10-17].

Schalten einer Anzeige

```
1   app.get('/getImage', (req, res) => {
2     const fileToHost = getRandomFile();
3     if(fileToHost){
4       const filePath = path
5           .join(__dirname, './uploads/' + fileToHost);
6       res.sendFile(filePath);
7
8       const index = getIndexOfFile(fileToHost)
9       blockchain.showAd(index).then(res => {
10        console.log('Ad Shown');
11      }).catch(err => {
12        console.log('no Funds left');
13      })}
14    else{
15      res.sendStatus(400);
16    }
17  });
```

Für das Versorgen des Frontends mit Anzeigen wird eine GET-Anfrage unter der Sub-Adresse */getImage* definiert und da keine Dateien gespeichert werden, ist Multer für diese Funktion nicht notwendig [1]. Mithilfe der Hilfsfunktion *getRandomFile()* wird eine zufällige Anzeige aus dem Ordner, in dem sich alle Uploads befinden, ausgewählt und wenn eine gefunden wurde, diese an das Frontend gesendet [2-6]. Anschließend wird mithilfe einer weiteren Hilfsfunktion der Index als Ganzzahl aus dem Dateinamen extrahiert

und als Input für die Funktion *showAd* mitgegeben [8-13]. Sollte keine Datei gefunden worden sein, wird ein Fehler an das Frontend zurückgegeben [15].

4.2.3 Smart-Contract

Der Smart-Contract ist dafür zuständig, beliebige Daten auf der Ethereum-Blockchain sowohl abzuspeichern, als auch abrufbar machen zu können. Für den Smart-Contract wird, die in Kapitel 3.1.3 vorgestellte Programmiersprache, Solidity verwendet. Im Gegensatz zu den anderen Komponenten bedarf es hier keiner zusätzlichen Bibliotheken und der Code ist auch simpel gehalten, um hohe Kosten durch Gas-Verbrauch zu vermeiden. Die folgenden Funktionen, die alle auf das Mapping *ads* zugreifen, sind jene, die von den anderen Komponenten aufgerufen werden.

Anpassen des Guthabens

```
1  function augmentAds(uint _id, uint _wei) payable public{
2    ads[_id].funds += _wei;
3  }
```

Die Funktion *augmentAds()* wird sowohl vom Backend, als auch Frontend in verschiedenen Kontexten aufgerufen. In beiden Fällen wird das Mapping an der Id angepasst, indem man das Guthaben um den gewünschten Betrag *_wei*, welcher als Parameter mitgegeben wird, erhöht.

Aufrufen einer Anzeige

```
1  function showAd(uint _id) public {
2    ads[_id].counter++;
3    ads[_id].funds -= 20000;
4  }
```

Die Structure *Ad* besteht nicht nur aus *funds*, sondern enthält auch einen Zähler, welcher dann in der Funktion *showAd()* erhöht wird, wenn das Backend dem Frontend eine Anzeige zur Verfügung stellt. Außerdem wird das Guthaben um einen manuell festgelegten Betrag reduziert. Diese Funktion stellt somit den Bereich des Vertrages dar, in dem die Kosten für das Schalten von Anzeigen festgelegt werden.

Abrufen der Schaltungen

```
1  function getCounter(uint _id) public view returns(uint){
2    return ads[_id].counter;
3  }
```

Die Funktion *getCounter()* gibt die Anzahl der Schaltungen für einen gegebenen Index wieder. Unternehmen könnten somit selbstständig überprüfen, wie oft ihre Anzeigen geschaltet wurden. Analog dazu könnte auch das übrige Guthaben für die Id abgefragt werden.

4.2.4 Provider und Deployment

Die Architektur der Webanwendung ist mit den bisher beschriebenen Komponenten nicht vollständig lauffähig. In ihrem jetzigen Zustand sind Front- und Backend nicht in der Lage, mit dem Smart-Contract, welcher sich noch nicht auf der Blockchain befindet, zu kommunizieren. Um dies zu korrigieren, sind die bereits genannten Provider notwendig, die sich nach dem Deployment des Smart-Contracts um die Kommunikation mit diesem kümmern.

Provider und Signer

Alle Funktionsaufrufe des Smart-Contracts werden durch Transaktionen ausgelöst, die von einem Teilnehmer des Netzwerkes, also einer Full-Node, signiert werden müssen. Um eine solche Node nicht selbst betreiben zu müssen, nimmt man die Dienste sogenannter *Provider* in Anspruch, welche das Senden von Transaktionen übernehmen. Das Front- und Backend verwenden zwei verschiedene Provider:

- *Metamask* im Frontend
- *Infuria* im Backend

Metamask ist eine Browser-Erweiterung, die unter *Google Chrome* und *Mozilla Firefox* frei erhältlich ist. Bei Metamask handelt es sich um eine Wallet-Software, die, nach Zustimmung des Users, eine Transaktion, welche von einer Webanwendung gefordert wird, signiert. Im Gegensatz dazu wird im Backend Infuria als Provider verwendet, denn dort kann nicht mit einem User-Input gearbeitet werden. Stattdessen gibt man als 'Eigentümer des Projekts' dem Provider den private Key und ermächtigt diesen, alle eintreffenden Transaktionen zu signieren. Die Wallet, mit der in beiden Fällen eine Transaktion signiert wird, nennt man *Signer*

Truffle

Bevor Funktionen eines Smart-Contracts aufgerufen werden können, muss dieser kompiliert und auf die Blockchain migriert werden. Das Framework *Truffle* bietet einem, ähnlich wie Angular, ein Kommandozeilen-Interface, welches beispielsweise mit dem Befehl *truffle init* eine Vorlage für einen neuen Smart-Contract erstellt. Außerdem wird ein Migrationsskript erzeugt, welches nach Belieben angepasst werden kann, um das Kompilieren und Deployment von mehreren Dateien gleichzeitig zu ermöglichen. In der erzeugten Config-Datei ist definiert, welche Netzwerke zur Verfügung stehen und welcher Signer für das Signieren der Deployment-Transaktion verwendet werden soll. Nach der Kompilierung wird eine JSON-Repräsentation des Smart-Contracts erzeugt, die neben der Adresse und anderen Informationen das *Application Binary Interface*, kurz *ABI*, enthält. Dies ist eine Sammlung aller Funktionssignaturen des Smart-Contracts, welches den Rahmen der Interaktion mit diesem definiert. Folgendermaßen kann anschließend eine bedienbare Instanz des Smart-Contracts erzeugt werden:

```
1  const contract = new ethers.Contract(contractJson.networks['3']
2                  .address, ABI, provider)
3  const signedContract = contract.connect(signer);
```

Es wird mithilfe von ethers eine Instanz der Klasse 'Contract' erzeugt. Der Konstruktor nimmt drei Parameter entgegen:

- Die Adresse des Smart-Contracts auf dem gewünschten Netzwerk
- Die ABI, welche die Signaturen aller Funktionen enthält
- Den Provider, der sich um die Kommunikation mit der Blockchain kümmert

Indem man den Contract mit einem Signer, also einer Wallet, verbindet, autorisiert man den Provider dazu, alle anfallenden Kosten für Transaktionen von der Wallet übernehmen zu lassen. Alle, in der ABI definierten, Funktionen sind aufrufbar und schlussendlich ist die gesamte Anwendung lauffähig.

4.3 Beantwortung der Forschungsfrage

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung der Kapitel

5.2 Diskussion der Ergebnisse

5.3 Ausblick

A Anhang

A.1 Anhang A

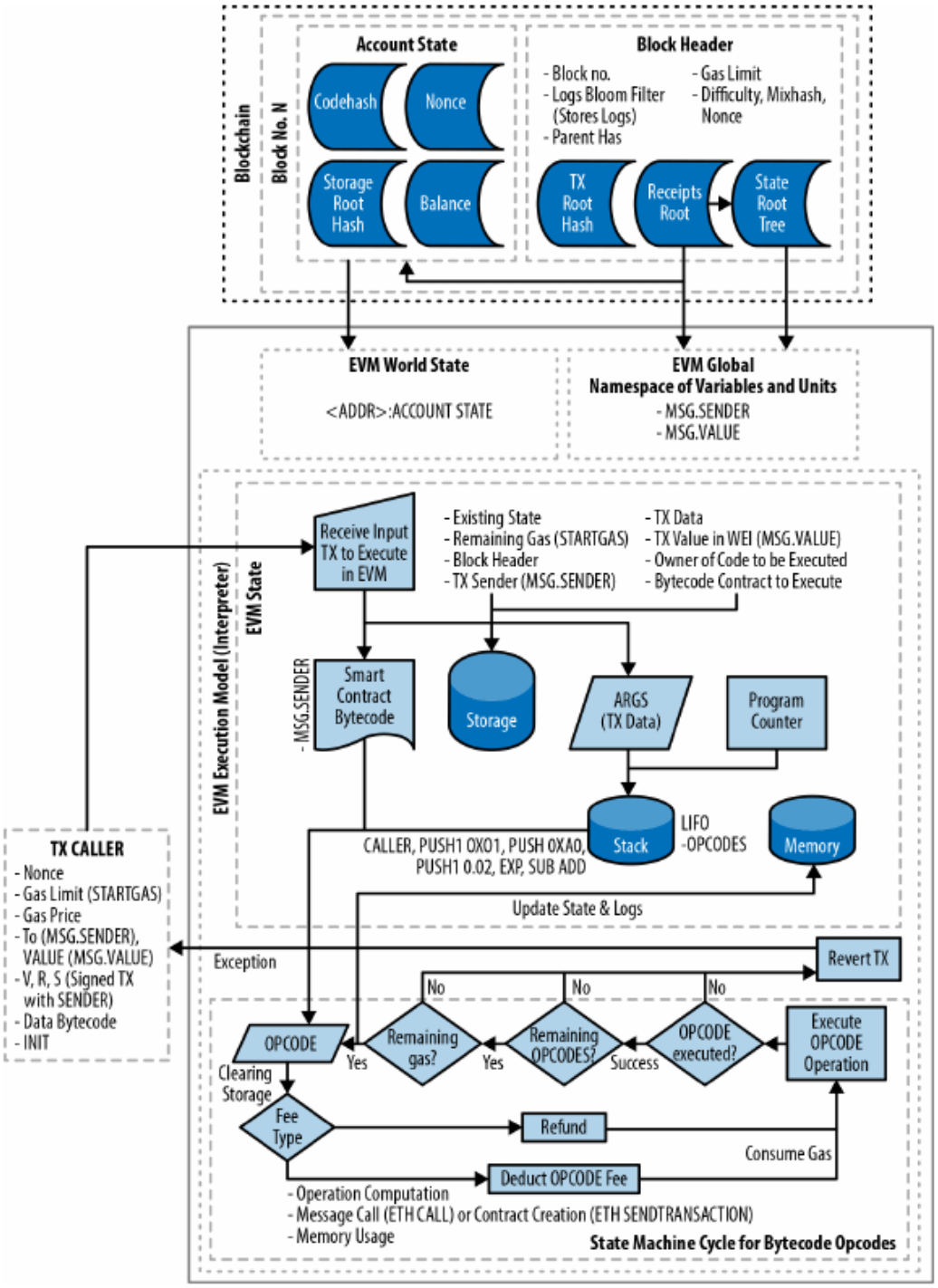

Abbildung A.1.: Ausführliche Darstellung der Ethereum Virtual Machine

Literaturverzeichnis

Antonopoulos, A. M. (2014). *Mastering Bitcoin: Unlocking Digital Crypto-Currencies* (Open Edition. Aufl.).

Antonopoulos, A. M. (2018). *Mastering Ethereum* (Open Edition. Aufl.).

Back, A. (2002). Hashcash - a denial of service counter-measure. http://http://www.hashcash.org/papers/hashcash.pdf. Letzter Zugriff am 23.5.2021.

Bundeskartellamt (2018). Online advertising.

Buterin, V. (2013a). Deterministic wallets, their advantages and their understated flaws. https://bitcoinmagazine.com/articles/deterministic-wallets-advantages-flaw-1385450276/. Letzter Zugriff am 21.02.2021.

Buterin, V. (2013b). A next generation smart contract & decentralized application platform. https://ethereum.org/en/whitepaper/. Letzter Zugriff am 12.4.2021.

Corbellini, A. (2015). Elliptic curve cryptography: finite fields and discrete logarithms.

Dai, W. (1998).

Dang, Q. H. (2015). Secure Hash Standard (SHS). http://dx.doi.org/10.6028/NIST.FIPS.180-4/. Letzter Zugriff am 14.02.2021.

Diffie, W., M. E. Hellman (1976). New directions in cryptography. https://ee.stanford.edu/~hellman/publications/24.pdf. Letzter Zugriff am 23.05.2021.

Google (2021). Adsense revenue share. https://support.google.com/adsense/answer/180195?hl=en. Letzter Zugriff am 12.5.2021.

Johnson, J. (2021). Global digital population as of january 2021. https://www.statista.com/statistics/617136/digital-population-worldwide/. Letzter Zugriff am 12.5.2021.

Landwehr, D. (2008). *MYTHOS ENIGMA*. transcript Verlag, Bielefeld.

Mankiw, N. G., M. P. Taylor (2018). *Grundzüge der Volkswirtschaftslehre* (7. Aufl.). Schäffer-Poeschel Verlag Stuttgart, Stuttgart.

Michaelson, G. (2020, Feb). Programming paradigms, turing completeness and computational thinking. *The Art, Science, and Engineering of Programming 4*(3).

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

Statista (2021). Online advertising revenue in the united states from 2000 to 2020. https://www.statista.com/statistics/183816/us-online-advertising-revenue-since-2000/. Letzter Zugriff am 12.5.2021.

Turing, A. M. (1936). On computable numbers with an application to the entscheidungsproblem.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf/. Letzter Zugriff am 20.4.2021.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang B.Sc. Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den _____ Unterschrift: _____