Transaktionen

Ture Claußen, 1531067, ture.claussen@stud.hs-hannover.de und Jannes Neemann, 1530893, jannes.neemann@stud.hs-hannover.de

Fakultät IV, Abteilung Informatik, Hochschule Hannover, Ricklinger Stadtweg 120, 30459 Hannover

Selbständigkeitserklärung

Mit der Abgabe der Ausarbeitung erklären wir, dass wir die eingereichte Seminar-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von uns angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben.

Hannover, den 23. April 2020

Zusammenfassung. Transaktionen verändern den Zustand der Ethereum-Blockchain.

Schlüsselwörter: Ethereum · Transaktionen · RLP · Gas · Propagation

Inhaltsverzeichnis

1 Einführung

Das Wort Transaktion stammt von dem lateinischen Wort *transigere* ab, welches im übertragenden Sinne mit 'durchführen', 'vollführen' oder 'abmachen' (Geschäft) übersetzt werden kann. [9] Dieser Wortsinn besteht auch weiterhin im technischen und wirtschaftlichen Bereich, jedoch gibt es noch spezifischere Abgrenzungen. In der Wirtschaft ist es ein Vorgang bei dem Waren und Forderungen ausgetauscht werden. [15, S. 18 f.] In der Informatik ist es im Zusammenhang mit Datenbanken eine unteilbare, *atomare*, Abfolge von Anweisungen, die einen Übergang von einem konsistenten Zustand in einen Anderen beschreibt. [16, S.520]

Ethereum ist ein "transaktionsbasierter Automat"(*transaction-based state machine*). Somit sind Transaktionen ein grundlegender Baustein von Ethereum im Allgemeinen und ihnen kommt eine ähnliche Bedeutung wie ACID Transaktionen bei. Der Automat speichert seinen Zuständ σ_t in der Blockchain, eine Transaktion T ist Argument der Zugstandsübergangsfunktion Υ, die von *externen Akteuren (EA)* angestoßen wird und diesen gespeicherten Zustand σ_t in einen neuen, gültigen Zustand σ_{t+1} überführen soll: $\sigma_{t+1} = \Upsilon(T, \sigma_t)$. Im Falle eines Konsens des Netzwerkes wird diese Zustandsveränderung durchgeführt beziehungsweise gespeichert.

Im Kontrast zu Kryptowährungen wie Bitcoin ist der Umfang des Automaten bzw. des Protokolls bei Ethereum deutlich geweitet, denn Zweck ist nicht nur die Schöpfung, Speicherung und der Austausch eines digitalen Zahlungsmittels [20], sondern eine allgemeine dezentrale Rechenmaschine, ein "Weltcomputer". [27, S. 1-4]

Im Folgenden betrachten wir eine beispielhafte Transaktion T_x von der Erzeugung, Signatur und Veröffentlichung, woran wir die technischen Spezifikationen von Transaktionen in Ethereum erläutern werden.

2 Struktur und technische Umsetzung einer Transaktion

Die Komponenten welche eine Transaktion in Ethereum ausmachen sind vergleichbar mit denen eines Briefes. Sie besitzen einen Empfänger sowie eine Frankierung, welche die Transportkosten zum Empfänger bezahlen. In einen Brief kann man z.B. Geld oder einen Text verschicken. Auch Transaktionen in Ethereum können Geld in Form von Ether und einen Text in Form von Nutzdaten verschicken. Im Weiteren wir die allgemeine Struktur und technische Umsetzung einer Transaktion vorgestellt.

2.1 Komponenten einer Transaktion

Transkationen, so auch unsere Transaktion T_x, enthalten laut ihrer offiziellen Definition [27, S. 4] folgende Datenfelder:

nonce: Ein Skalar welcher gleich der Anzahl der vom EOA versendeten Transaktionen ist. Der Nutzen wird in 3.1 erläutert.
gasPrice: Ein Skalar der angibt, wie viel Wei man pro Einheit *Gas* bezahlt, die bei der Gesamtheit aller Berechnungen die während der Ausführung der Transaktion anfallen (S. 3.2)
gasLimit: Ein Skalar der die maximal Anzahl an *Gas* angibt, die während der Ausführung der Transaktion verbraucht werden darf. Dieser Betrag muss im Voraus bezahlt werden.
to: Die 160-Bit Adresse des Empfängers.
value: Skalar der die Menge Wei angibt, die der Empfänger erhält.
v,r,s: Komponenten der ECDSA-Signatur (S. 3.4), um den Sender der Transaktion zu bestimmen
init: Ein Byte-Array unbegrenzter Länge, welches nur bei einer Kontrakterzeugung verwendet wird und den kompilierten Sourcecode des Kontrakts enthält
data: Ein Byte-Array unbegrenzter Länge, welches die Nutzdaten des Kontrakts enthält

2.2 Typen von Transaktionen

Es gibt genau zwei Typen von Transaktion. Transaktionen die eine Nachricht von einem Account[1] zu einem anderen überträgt („message calls“ [27, S. 4]) oder Transaktionen die einen neuen Kontrakt erzeugen („contract creation“ [27, S. 4])). Mit Nachricht ist dabei der Inhalt von den Feldern *value* und *data* gemeint.

Bei massage call Transaktionen enthält das *to* Feld die öffentliche Adresse eines EOA oder eines Kontrakts. Zusätzlich besteht die Option *value* und *data*

[1] Mit Account ist hier ein EOA oder ein Kontrakt gemeint

zusetzen. Speziell ist das *data* Feld von Bedeutung, wenn das Transaktionsziel ein Kontrakt ist, denn in diesem Feld wird der konkrete Funktionsaufruf in Bytecode gespeichert und wird von dem Kontrakt ausgeführt.

Die Besonderheit bei contract creation Transaktionen ist, dass die Empfängeradresse die Nulladresse (*0x0*) ist. Diese Adresse ist keinem Account zugewiesen und dient ausschließlich als „kontrakterzeugungs Adresse“ [11].

Ein spezieller Typ von Transkation ist eine interne Transaktion. Diese treten nur ausgehend von einem Kontrakt aus auf. Sie werden auch nicht in der Blockchain aufgefasst. Somit kann eine Transkation die an einen Kontrakt gerichtet ist, eine Funktion aufrufen, welche dem Sender einen Etherbetrag zurücksendet. Diese eigentlich Transaktion wird als interne Transaktion gewertet und nur der Funktionsaufruf wird in der Blockchain dokumentiert.

2.3 Serialisierung

Da Ethereum ein Weltcomputer ist und Daten somit über die ganze Welt verschickt werden, müssen diese kompakt, effizient und einheitlich verschickt werden. Dabei wird das Kodierungsverfahren *Recursive Length Prefix (RLP)* verwendet. Alle serialisierten Daten in Ethereum sind Listen von Bytes [27, S. 3]. Auch die Daten einer Transaktion werden mit Hilfe von RLP in eine Liste von Bytes serialisiert und wieder deserialisiert. Bei der RLP Kodierung handelt es sich nur um ein Verfahren um Struktur zu serialisieren. Das heißt die Methode nimmt nur ein so genanntes „Item“ als Parameter entgegen. Dieses Item ist entweder ein String, welcher in ein Byte-Array konvertiert wird, oder eine Liste von Items. Relevant ist für die Methode nur die Länge des Items. Je nach Fall werden dabei unterschiedliche Regeln definiert:

1. Item ist eine Zeichenkette (Byte-Array):
   - Besteht dieses nur aus einem Byte mit einem Wert kleiner als 128 (`0x7f`), ist das Byte ihre eigene RLP Repräsentation
   - Enthält das Byte-Array weniger als 56 Byte, ist die RLP Repräsentation der Inhalt dieses mit einem Präfix von `0x80` (128) plus die Länge des Arrays. Beispiel: „Ethereum“ ⇒ `[0x85, 'E', 't', 'h', 'e', 'r']` bzw. mit ASCII Kodierung : `[0x85, 0x45, 0x74, 0x68, 0x65, 0x72]`
   - Ist das Byte-Array größer als 55 Byte wird ein Präfix aus mehreren Bestandteilen verwendet. Zum einen `0xb7` plus die Anzahl der Bytes die benötigt werden, um die Länge des String darzustellen. Gefolgt von der Länge des Strings im Big-Endian Format und dem Inhalt des Byte-Arrays. So ergibt sich für ein 2048-Byte langes Byte-Array folgender Präfix: `[0xb9, 0x80, 0x00]`. 2048 entsprechen in Hexadezimal `0x800` somit werden zwei Bytes (`0x80` und `0x00`) benötigt, um die Länge des Bytes darzustellen. Somit erhalten wir $\texttt{0xb7} + 2 = \texttt{0xb9}$.
2. Item ist eine (verschachtelte) Liste von Items:
   - Ist die Gesamtlänge alle in der Liste enthaltenen Items mit ihrer jeweiligen RLP Repräsentation 0-55 Bytes lang, so wird der Präfix `0xc0`

plus die Länge der konkatenierten Liste der RLP Repräsentation gesetzt. Anschließend folgt die Liste selbst. So wäre die Kodierung der Liste `[Ëther", "Wei"] [0xca, 0x85, 'E', 't', 'h', 'e', 'r', 0x83, 'W', 'e', 'i']` bzw. inklusive ASCII-Kodierung `[0xca, 0x85, 0x45, 0x74, 0x68, 0x65, 0x72, 0x83, 0x57, 0x65, 0x69]`. Das zweite bis siebte Byte ist dabei die RLP Repräsentation von „Ether“ und die Bytes acht bis elf die von „Wei“. Somit ergibt sich eine Länge von 10 Byte, somit lautet der Präfix `0xca`

– Ab einer Gesamtlänge von 56 Bytes wird der Präfix `0x7f` plus die Anzahl der Bytes die benötigt werden, um die Länge der Liste darzustellen. Danach folgt die Länge der Liste mit der konkatenierten Liste von RLP Repräsentation

Das Item darf nicht länger als 2^{64} Bytes sein, da sonst die Länge des Präfix in allen Fällen länger als 255 ist und somit nicht in einem Byte dargestellt werden kann.

3 Aufbau einer Transaktion

3.1 Nonce

Die Nonce wurde nicht von Ethereum eingeführt, sondern kommt aus dem Bereich der Kryptographie. Eine Nonce ist dort laut Definition eine willkürliche Nummer, die nur einmal in einer kryptographischen Kommunikation verwendet wird. Dabei handelt es sich meistens um eine zufällig oder pseudo-zufällig generierte Zahl. Mit der die Einmaligkeit der Kommunikation gesichert wird.

In Ethereum-Transaktionen ist die Nonce eine Zahl, welche bei der Accounterstellung den Wert Null hat und bei jeder erfolgreichen Transaktion[2] um eins inkrementiert wird. Dieser Wert wird dabei nicht explizit im Account in der Blockchain gespeichert, sondern dynamisch, indem die Anzahl der erfolgreichen Transaktionen gespeichert wird [11, S.101].

Mit der Nonce werden sogenannte ”Replay attacks”verhindert. Im Rahmen von Ethereum gesprochen wird so verhindert, dass die selbe Transkation mehrmals ausgeführt werden kann. Da Transaktionen in der Blockchain gespeichert werden und alle Daten zu dieser Transkation eingesehen werden können, wäre es ohne die Nonce möglich, dass eine Unbeteiligter der Transkation diese unbegrenzt oft wiederholen kann, ohne die Zustimmung des Absenders zu haben. Da die Transkation jedoch schon einmal abgeschlossen ist, entspricht die des Absenders nicht mehr der der Transaktion, somit kann die Transkation nicht widerholt werden. Das gleiche gilt, wenn der Absender die Transkation wiederholen möchte.

Des Weiteren dient die Nonce auch der Transaktionsabwicklung innerhalb des Netzwerks. Werden mehrere Transaktionen von einem Account versendet, kommen diese meistens in unterschiedlicher Reihenfolge bei den Nodes an. So

[2] Eine Transaktion ist erfolgreich, wenn sie in einem Block der Blockchain aufgenommen wurde

ist nicht sichergestellt, dass eine Transaktion die eine höhere Priorität hat, auch als erste verarbeitet wird. Mit der Nonce kann dies jedoch realisiert werden. So vergleicht das Netzwerk die Nonce, die mit der Transkation gesendet wird, mit der Nonce des Account. Stimmen diese überein, so wird die Transkation sofort verarbeitet. Ist die Nonce der Transaktion größer als die erwartet, landet die Transkation im *Mempool*, in der sich alle noch nicht verarbeiteten Transkationen befinden. Ist die Nonce des Accounts zum Beispiel 2 und die der Transkation 5, so geht der Node davon aus, dass die Transaktionen mit den noch fehlenden Noncen sich verspäten. Somit bleibt die Transaktion solange im Pool, bis die Transaktion mit den Nonce 2, 3 und 4 im Netzwerk registriert wurden. Somit kann eine Priorisierung von Transaktionen durch eine in der Priorität absteigende Transaktionsreihenfolge sichergestellt werden.

3.2 Gas

Gas ist ein zentraler konzeptioneller Lösungsansatz im Rahmen von Ethereum. Da Ethereum turing-vollständig ist [27, S. 1], ergibt sich unter anderem das sogenannte "Halteproblem". Dieses besagt, dass im Voraus nicht vorhergesagt werden kann, ob das Programm einer Turing-Maschine jemals zu einem Ende kommt. [14, S.70] Um die Funktionalität des Netzwerks zu gewährleisten, wird die Laufzeit einer jeden Zustandsveränderung der Blockchain, sprich Transaktion, durch Gas begrenzt.

Gas ist eine eigenständige Währung innerhalb von Ethereum, dessen Einheit einen Rechenschritt in der EVM bemisst [18, S. 9:3], wobei für jeden Opcode die Kosten in Gas spezifiziert werden. [27, S. 25 ff.] Gas ist also eine Gebühr für Rechnenaufwand. Zusätzlich werden auch Kosten für die Nutzung von persistentem Speicher miteinbezogen. Es gilt sogar das Inverse: Wird durch eine Transaktion persistenter Speicher freigegeben, werden Rabatte gewährt.

Die maximale Gebühr einer Transaktion wird durch die Kombination der Datenfelder *gasPrice* und *gasLimit* angegeben. Die resultierende Gebühr *gasPrice*× *gasLimit* wird bei Erstellung der Transaktion in voller Höhe vom Konto abgezogen. Nach Bestätigung der Transaktion wird nicht genutztes Gas zu dem angegebenen Preis in Ethereum zurückerstattet.

Somit gilt es nun im Voraus abzuschätzen wie hoch der Rechenaufwand für unsere Beispieltransaktion T_x sein wird. Je mehr Ressourcen des Weltcomputers sie in Anspruch nimmt, desto höher die Gebühr. Gerade wegen des Halteproblems kann dies aber nur grob vorgenommen werden, eine robuste Programmierung von *Smart Contracts* ist essentiell. Ein erster Anhaltepunkt dafür sind zunächst die intrinsischen Kosten einer Transaktion. Das ist der Overhead der allein durch die Transaktion und deren Inhalt betsteht. Diese intrinsischen Kosten g_0 lassen sich mit auf Basis folgender Grundlage berechnen.

$$g_0 \equiv \sum_{i \in T_i, T_d} \begin{cases} G_{datazero} \text{ if } i = 0 \\ G_{txdatanonzero} \text{ otherwise} \end{cases} + \begin{cases} G_{txcreate} \text{ if } T_t = \emptyset \\ 0 \text{ otherwise} \end{cases} + G_{transaction}$$

Also steigen die Kosten einer Transaktion mit der Größe des Feldes *data* an und $G_{transaction}$ bestimmt den Basiswert an Gas für eine Transaktion, welcher sich im Jahr 2020 auf 21000 beläuft. Generell sollte das *gasLimit* tendenziell zu hoch angelegt sein, da Transaktionen mit unzureichendem Gas einfach abgebrochen werden *(out-of-gas Exception)*. In diesem Fall wird keine der begonnenen Veränderungen am Zustand gespeichert. Durch Einsetzen für g_0 ergeben sich für die Beispieltransaktion T_x intrinsische Kosten von: X TODO calculate

Preis und Latenz Gas kann bewusst nur mit Ether erworben werden, da die Gas-Preise möglichst unabhändig von den Preisschwankungen (von Ether) sein sollen. Der *gasPrice* kann frei gesetzt werden, auch ein Wert von 0 ist gültig. Ein Richtwert für den Wert lässt sich durch Werkzeuge wie ETH Gas Station ermitteln, welche vergangene Transaktionen im *Ledger* betrachten und daraus Richtwerte ermitteln. Für T_x empfiehlt sich ein *gasPrice* von 9 GWei laut ETH Gas Station. Am 20.04.2020 aktzeptieren ungefähr 84% der letzten 200 Blöcke diesen Preis, sodass sich für unsere Transaktion T_x Kosten in Höhe von TODO CALC $\times 9\text{Gwei} = X$ ergeben.

Dort wird auch ein Umstand kenntlich, denn die Höhe des Gas-Preises scheint maßgeblich über die Latenz zu entscheiden, also die Zeit bzw. Zahl der Blöcke zwischen Veroeffentlichung einer Transaktion und ihrer Inkludierung in einem Block. Übersteigt der *gasPrice* das Mittel der anderen Transaktionen im *mempool* so steigt die Wahrscheinlichkeit in nächsten Block bearbeitet zu werden. Diese Korrelation schwindet allerdings, sobald die Durchsatzfähigkeit des Netzwerkes erreicht ist. [22, S. 30 f.]

Zusätzlich kann das *gasLimit* H_1 eines Blocks nur durch die Miner nach erfolgreichem Schürfen eines Blockes um maximal $\frac{P(H)_{H1}}{1024}$ des alten Limits $P(H)_{H1}$ erhöht oder verringert werden. Dies soll eine Zentralisierung der Rechenleistung auf wenige, große Miner verhindern. Gleichzeitig limitiert dies die Fähigkeit viele Transaktionen in einem kurzen Zeitintervall zu verarbeiten oder dynamisch auf eine höhere Last zu reagieren.

Unter Betrachtung aller Transaktionen im Zeitraum vom 01.03.2020 00:00:17 UTC (Block 9581792) bis 31.03.2020 23:59:57 UTC (Block 9782601) mit dem Python Werkzeug *ethereum-etl* [10] ergibt sich aktuell folgender durchschnittlicher Durchsatz T_{max} pro Block: [21]

$$T_{max} = \frac{blockGasLimit}{transactionMedianGas} = \frac{9817880}{80000} = 122.72$$

Bei kurzzeitig stark erhöhter Anzahl an Transaktionen wie beispielsweise bei einem *ICO (initial coin offering)*, werden teilweise um ein Vielfaches höhere Transaktionskosten gezahlt, um möglichst schnellen Zugriff auf die Wertanlagen zu erhalten. [18, S. 9:6 f.] Im Betrachteten Zeitraum war dies zum Beispiel am 13.03.2020 der Fall (s. 1). Zu diessem Zeitpunkt wurde teilweise 800 GWei pro Einheit Gas gezahlt. Der Angriff ist zeitlich stark mit einem *DOS-Angriff* auf die Börse Bitmex korreliert. Ob dies tatsächlich die Ursache für den extremen Anstieg der Netzwerkaktivität ist, wird allerdings im folgenden nicht weiter analysiert. [13]

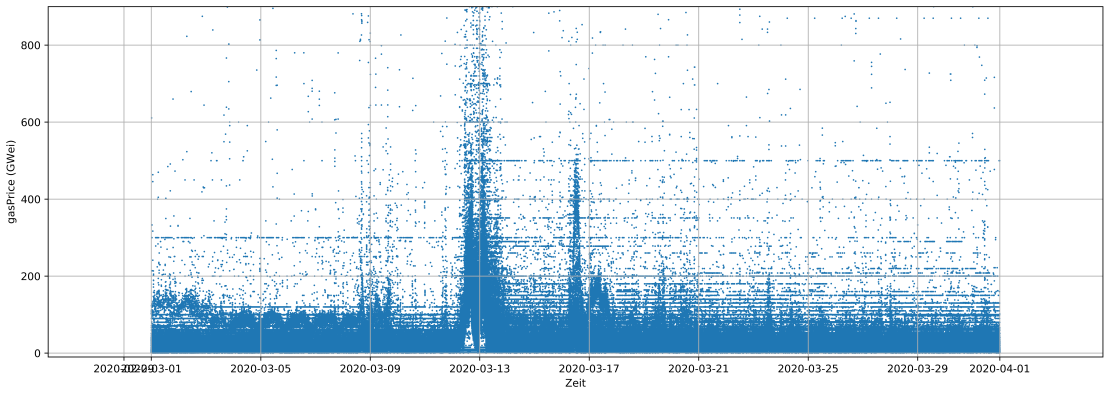

Abb. 1. gasPrice nach Tag im Monat März [21]

Bei Betrachtung der Verteilung der Anazahl Transaktionen pro Block 2 zeigt sich ein weiteres Problem. Zunächst verteilt sich das Gros der Transaktionen ungefähr um den zuvor berechneten Durchsatz T_{max}. Jedoch gibt es einen beträchtlichen Anteil an Transaktionen, der fast keine oder sehr wenige Transaktionen enthalten und gleichzeitig nur einen Bruchteil des *blockGasLimit* ausschöpfen, wobei sogar 3,4% der Blöcke komplett leer sind. [21] Es können mehrere Vermutungen angestellt werden, wo der Grund dafür liegt.

Alle Miner stehen unter einem wirtschaftlichen Druck. Es werden beträchtliche Ressourcen investiert, eine Auszahlung der aufgewendeten Rechenleistung gibt es nur, wenn ein Block erfolgreich geschürft wird. Wenn die Lösung für den nächsten Block gefunden wurde, muss diese noch über das Netzwerk an die anderen Nodes veröffentlicht werden. Je mehr Transaktionen im Block vom Miner gespeichert werden, desto länger dauert die Verifizierung der Transaktionen. Beim sogenannten *SPV-Minig* wird sogar komplett auf die Verifizierung verzichtet. [25] Hier erhöhen mehr Transaktionen die Gefahr eines *double-spend* und die Ungültigkeit des Blockes. Außderdem verbrauchen mehr Transaktionen mehr Speicher (ergo Bandbreite) benötigt der Block. [23]

Alle Vermutungen deuten in die gleiche Richtung: Je mehr Zeit verbraucht wird, steigt proportional dazu das Risiko, dass keine Belohnung ausgezahlt wird, da ein anderer Miner seine Lösung schneller veröffentlicht.

Anreiz und Spieletheorie Es lässt sich postulieren, dass beide Probleme auf das Anreizsystem von Ethereum zurückzuführen sind. Generell stellt sich für die Teilnehmer des Netzwerkes die Frage, warum sie eigene Ressourcen aufwenden sollten, um das Netzwerk zu ermöglichen. Nun kann natürlich davon ausgangen werden, dass alle altruistisch motiviert sind und aus Idealismus ihren Beitrag leisten. Ethereum geht aber von einer anderen Prämisse aus. Das Feld der *Cryptoe-*

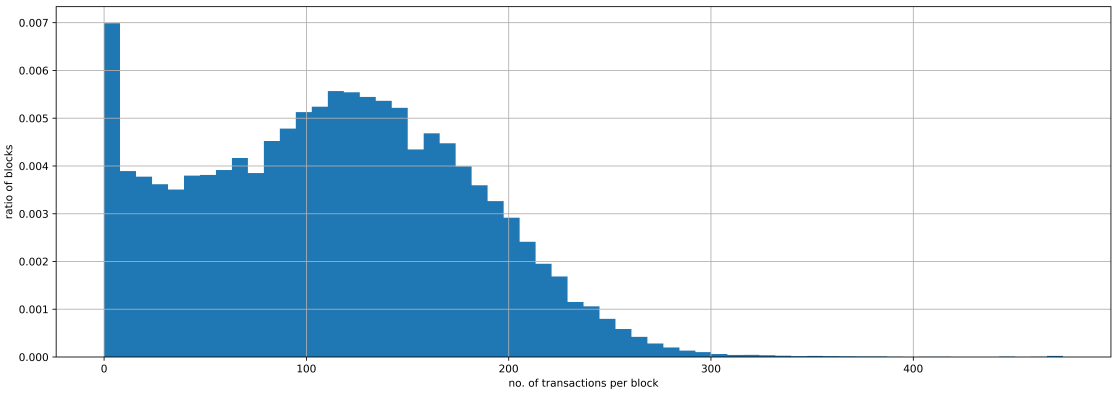

Abb. 2. Verteilung der Zahl an Transaktionen pro Block (60 konstante Klassen) [21]

conomics beschreibt wie ökonomische Anreize genutzt werden, um die Funktion und die Korrektheit von Kryptowährungen zu garantieren. Die Modellierung von Ethereum und seinem Anreizsystem basiert auf spieletheoretischen Ansätzen. [1]

In der Spieletheorie wird davon ausgegangen, dass Entscheidungsträger rational sind und sie die Nützlichkeit (*utility payoff*) ihrer Entscheidungen, unter Berüchksichtigung ihres Zustandes t und der möglichen Belohnungen $x \in X$, maximieren wollen. Unter dieser Annahme muss eine Kryptowährung also Anreize schaffen, die für den Fortbestand des Systems zuträglich sind und gleichzeitig den *utility payoff* Nützlichkeitsfunktion $u(x, t)$ möglichst maximieren. [19, S. 2 ff.]

Das Protokoll von Ethereum bildet den Rahmen des Spiels und seine Regeln. Überträgt man zum Beispiel genannte Problem der leeren Blöcke auf dieses Modell, ergibt sich eine mögliche Erklärung des Verhaltens. Die Rentabilität von Mining lässt sich unter Berüchksichtigung der aktuellen *block rewards*, *block difficulty*, etc. sehr gut errechnen. [2] Nun kann sich durch Erfahrungswerte des rationalen Entscheidungsträgers herausstellen, dass die Nützlichkeit höher liegt, wenn keine Transaktionen in Blöcken inkludiert werden. Als Resultat sind rein wirtschaftlich motivierte deutlich eher dazu geneigt leere Blöcke zu schürfen, obwohl dies jeglichem Sinn des Netzwerkes widerspricht.

Daran zeigt sich wie schwierig es ist, ein wohl balancierts Anreizsystem zu entwerfen, im Anbetracht der Komplexität der Realität. Ein angedachter Wechsel zu *proof of stake* birgt das Potential derartige Probleme zu beheben, aber gleichzeitig wird es unweigerlich neue unerwartete Nebenwirkungen mit sich bringen. [5]

3.3 Value und Data

Wie in 2.1 schon vorgestellt, enthalten das *value*- und *data*-Feld die eigentliche Nutzlast einer Transaktion. Dabei enthält das *value*-Feld ausschließlich den Betrag an Wei, der an die Empfängeradresse gesendet werden soll und das *data*-Feld enthält die Nachricht. Eine Transkation die ein *value*-Feld enthält, wird dabei auch Zahlung bzw. *payment* genannt. Das *data*-Feld ist ein Aufruf bzw. *invocation* [11, S.108]. Eine Zahlung zwischen zwei EOAs ist dabei eine einfache Zustandsänderung der EVM und Übertragung des Etherbetrags in Weis auf den empfangenden Account. Enthält diese Transkation Daten im *data*-Feld so wird diese von der Blockchain ignoriert [27, S.10]. So werden diese auch von der eigenen Wallet ignoriert und nur angezeigt.

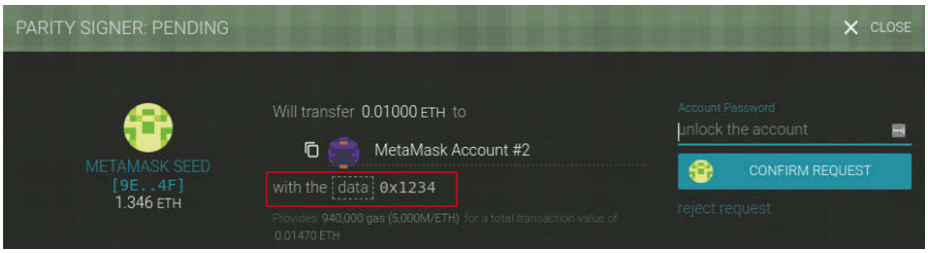

Abb. 3. Beispieltranskation an EOA mit gefülltem *data*-Feld [11, S.109]

Relevant wird der Inhalt, wenn es sich bei der *to* Adresse um einen Kontrakt handelt. Der Inhalt wird dann von der EVM als ein Funktionsaufruf interpretiert. Ein Kontrakt enthält beispielsweise folgende Funktion:
`function getTokenValueFromWei(uint256) public return (uint ret) {...}`
Diese Funktion ist Teil eines Token-Kontrakt und gibt den Wert des übergebenen Weis in diesem Token zurück. Relevant wird der Inhalt, wenn es sich bei der *to* Adresse um einen Kontrakt handelt. Der Inhalt wird dann von der EVM als ein Funktionsaufruf interpretiert. Unsere Transaktion T_x soll die Funktion

```
function deposit(string _depositReason) public payable {
  balances[msg.sender] += msg.value;
  reasons[msg.sender].push(_depositReason);
}
```

unseres Kontrakts aufrufen. Diese Funktion fügt den kontraktinternen Konto dem im *value*-Feld übergeben Wert hinzu. Dabei muss als Parameter der Einzahlungsgrund genannt werden.

Damit diese Funktion aufgerufen werden kann, muss der Funktionsaufruf der Spezifikation des Contract Application Binary Interface (ABI) entsprechen. Das heißt der endgültige Inhalt setzt sich im allgemeinen aus dem Funktionsselektor und den Funktionsargumenten zusammen. Der Funktionsselektor teilt dem

Kontrakt mit welche Funktion er ausführen soll und entspricht den ersten vier Bytes des Keccak-256-Hash, welches das meist genutzte Hash-Verfahren in Ethereum ist, des Funktionsprototypen. Laut ABI Spezifikation setzt sich der Funktionsprototyp aus dem Namen der Funktion und in Klammern folgend die einzelnen Parametertypen. Der Rückgabetyp einer Funktion ist nicht Teil des Funktionsprototyps. Daraus resultiert folgender Prototyp für unsere Funktion: `deposit(string)`. Dessen vollständiger Keccak256-Hash entspricht:

```
a26e11860cdb80ecca46e4f433c3c9533f6d37cdf0f6eb16343556cfdbcf47ec
```

Somit entspricht `0xa26e1186` dem ABI konformen Funktionsselektor.
Unser Funktionsaufruf soll 1000000000000000000 Wei (entspricht einem Ether) auf das Konto einzahlen. Als Parameter übergeben wir „Einzahlung“. Um einen String ABI konform zu kodieren müssen wir den Offset angeben, ab dem der Inhalt unseres Parameter startet, konkatiniert mit der Länge des Strings und beides nach links auf 32 Byte mit Paddingsbytes aufgefüllt. Danach folgt der String in UTF-8 kodiert, welcher nach rechts auf 32 Bytes aufgefüllt wurde. In unserem Fall müssen wir 32 Zeichen überspringen (`0x20`). Der String ist 10 (`0xa`) Zeichen lang. Somit lautet die Kodierung unserer Funktionsargumente wie folgt:

```
0x0000000000000000000000000000000000000000000000000000000000000020 \
000000000000000000000000000000000000000000000000000000000000000a \
45696e7a61686c756e6700000000000000000000000000000000000000000000
```

Unsere Nutzlast, welche wir im *data*-Feld nun eintragen müssen, erhalten wir aus der Konkatenation beider Kodierungen:

```
                                0xa26e1186 \
0000000000000000000000000000000000000000000000000000000000000020 \
000000000000000000000000000000000000000000000000000000000000000a \
45696e7a61686c756e6700000000000000000000000000000000000000000000
```

Damit unser Kontrakt auch die 1 Ehter erhält müssen wir diese im *value*-Feld eintragen. Unsere Funktion besitzt das Schlüsselwort `payable`, dieses wird bei Kontrakten verwendet um anzuzeigen, dass diese Funktion Ether annehmen kann. Dabei muss die im *data*-Feld aufgerufene Funktion nicht zwingend auch `payable` deklariert sein. Die EVM such in dem Fall alle Funktionen durch, bis sie eine so deklarierte Funktion findet. Wenn keine Funktion `payable` deklariert wurde, gibt es meistens eine sogenannte Fallback-Funktion, welche keinen Funktionsnamen besitzt und alleinig dazu dient Ether entgegen zu nehmen und diese dem Kontrakt gut zuschreiben. Ist auch diese nicht definiert, wirft die EVM eine Exception und die Transaktion wird abgebrochen. Man kann jedoch über das *data*-Feld keinen Ether an den Kontrakt übergeben. Dies ist ausschließlich über das *value*-Feld möglich. Damit der Kontrakt dieses Ether annimmt, muss die Funktion genau wie unsere Funktion mit dem Schlüsselwort `payable` deklariert sein. Akzeptiert die aufgerufene Funkntion kein Ether, so wird die sogenannte Fallback-Funktion aufgerufen, die den übergenenen Etherbetrag auf das Konto des Kontrakts gut schreibt. Ist auch diese nicht definiert, wird eine Exception geworfen und die Transaktion abgebrochen.

Die letzte mögliche Kombination ist, wenn sowohl das *value*- und *data*-Feld leer sind. Dies ist ebenfalls eine gültige Transkation würde. Diese erfüllt jedoch keinen besonderen Zweck außer der Verwendung des bezahlten Gas und somit nur einer Senkung des eigenen Kontostands.

3.4 Signatur

Die Signatur einer Transaktion belegt den Besitz eines Schlüssels, der aktuell die *Authenzität* und die *Integrität* der Nachricht beweisen kann. Dies basiert auf den Eigenschaften von *Trapdoor-Funktionen* im Allgemeinen und von asymmetrischer Verschlüsselung im Speziellen. Ohne Besitz des privaten Schlüssels ist es sehr leicht die Echtheit einer Nachricht zu prüfen, allerdings extrem schwierig eine gültige Nachricht zu generieren. [24]

Für eine gültige Transaktion in Ethereum braucht es einen zufälligen privaten Schlüssel k, also eine 256 bit große Zahl, aus dem dann wiederrum ein 512 bit großer öffentlicher Schlüssel K generiert wird. Mit dem keccak256-Hash von K kann daraus die Ethereum Addresse abgeleitet werden. Mit der Erzeugung eines privaten Schlüssels gelangt man also Kontrolle über einen Ethereum Account. Aufgrund der Größe des Schlüssels ist die Kollisionswahrscheinlichkeit zu vernachlässigen.

Um die Transaktion T_x mit einer Signatur Sig zu versehen, verwenden wir alle bisher genannten Datenfelder der Transaktion, also *nonce*, *gasPrice*, *gasLimit*, *to*, *value* und *data*, in RLP codierter Form. Nach dem DAO Hack und dem Fork zu Ethereum Classic wurde eine Erweiterung dieser sechs Felder in EIP-155 [3] um eine *Chain ID*, und $r = 0$ und $s = 0$ vorgeschlagen und in *Spurius Dragon* umgesetzt. Dies verhindert etwaige *Replay-Attacken*, bei denen Transaktionen über verschiedene Blockchainnetze gültig sind. [17, S. 138] Hier zeigt sich auch eine weitere Rolle des *nonce*, denn alle anderen Werte könnten potentiell identisch sein, sodass die resultierende Signatur auch identisch wäre. So ließe sich die Transaktion ungewollt wiederholen.

Der gesamte Korpus der Transaktion m wird dann in gehashter Form Argument des Signierungsalgorithmus zusammen mit dem privaten Schlüssel k.

$$\text{Sig} = F_{sig}(F_{keccak256}(m), k)$$

ECDSA Die resultierende Signatur Sig besteht aus den Komponenten r, s und dem Signaturpräfixwert v. Diese drei Komponenten sind Teil einer spezielle Form des *Digital Signature Algorithm*, die Elliptische-Kurven-Kryptographie verwendet. Der ursprüngliche Ansatz von *DSA* basiert auf der Annahme eines naiven Algorithmus, einer *Trapdoor-Funktion*, die auf dem großen Aufwandsunterschied zwischen Faktorisierung und Multiplikation beruht. Durch Verbesserung des ursprünglich zugrunde gelegten Algorithmus konnte dieser Aufwandsunterschied verkleinert werden, was immer größere Schlüssel erfordert. Bei *ECDSA* für das "Problem des diskreten Logarithmus in elliptischen Kurven"wurde noch kein bessere Ansatz als die naive Variante gefunden, sodass deutlich kleinere Schlüsselgrößen möglich sind.

Die Komponente r repräsentiert einen öffentlichen Punkt auf der elliptischen Kurve mit dem der s-Wert mit Hilfe des öffentlichen Schlüssel K verifiziert werden kann. Die Wert von v ist doppelt belegt. Einerseits gibt er eine weitere Referenz auf *Chain-ID*, andererseits kann mit ihm der öffentliche Schlüssel K des Senders schneller rekonstruiert werden, denn Transaktionnen enthalten kein

from Feld im klassischen Sinne. Wenden wir den Signaturalgorithmus auf die Transaktion T_x an, so erhalten wir folgende Komponenten, die dem Transaktionskörper angehängt werden: [11, S. 114 ff.]

v: 26
r: dade772f31d20b4ed1c7f63ae035c0cc83fd7b786ca9339eb01763138877a6d4
s: 13e16f7a55d261e504e27ea4fecc174a1a46c87d804b9a3917aebde665c1ddb1

Multisignaturen Eine spezielle Form der Signatur sind Multisignaturen. Bei einem *multi-signature* Schema muss eine Nachricht t von n aus m Entiäten unterzeichnet werden, wobei $m \geq n > 1$. [12, S. 2] Dies erlaubt es beispielsweise eine Absicherung gegenüber nicht vertrauenswürdigen Parteien zu schaffen, gemeinschaftlich ein Wallet zu verwalten oder gegen den Verlust eines privaten Schlüssels abzusichern. Während bei Bitcoin dies durch das Protokoll selbst ermöglicht wird [8], bedarf es bei Ethereum die Verwendung eines Contracts [6]

4 Transaktionsabwicklung

Wir haben nun alle notwendigen Konzepte und Inhalte einer Ethereumtranskation kennengelernt. Im weiteren wird vorgestellt, wie unsere Transkation T_x im Netzt verteilt wird und diese in der Blockchain aufgenommen wird. Damit unsere Transaktion versendet werden kann, werden unsere Transaktionsdaten mit dem privaten Schlüssel des sendenden Accounts signiert[3] und anschließend mit RLP kodiert [11].

4.1 Propagation

Bevor die Transaktion über das Ethereumnetzwerk verbreitet wird, überprüft der lokale Node (oder der Node, welcher MetaMask verwendet) ob die Transaktion wirklich von der eigenen Adresse stammt. Ist dies der Fall wird die Transkation über das P2P-Netwerk versendet. Ein Node ist mit mindestens 13 weiteren Nodes verbunden [11, S.123]. Jeder dieser erhält die Transkation und validiert dieser. Wenn dies erfolgreich ist sendet der jeweilige Node die Transaktion an seine Nachbarn weiter [11, S.123]. So wird erreicht, dass die Transkation sehr schnell bei jedem Node im Ethereumnetzwerk angekommen ist. Die Transaktion erreicht somit auch sogenannte Miner Nodes. Diese Nodes speichern unsere Transaktion in ihrem *Mempool*. Abhängig von unserer Platzierung in dieser Liste, wird die Transaktion an einem Zeitpunkt in einen Block aufgenommen. Sobald der der Block geschürft wurde, das heißt der *Proof of Work* gefunden wurde, findet eine Zustandsänderung des Zustandsautomat statt. Das heißt, die Funktion des Kontrakts wird von der EVM ausgeführt und das Ehter von dem Senderaccount abgebucht und dem Konto des Kontrakts gutgeschrieben. Schließlich kann man unsere Transaktion in der Blockchain wiederfinden:

[3] Es wird angenommen, dass ein eigener Node betrieben wird. Werden Anwendungen wie MetaMask verwendet übernimmt diese die Signierung

4.2 Speicherung

Die Speicherung und Ausführung der Transaktion lässt sich mit Hilfe der Felder *transactionRoot*, *receiptRoot* und *logsBloom* des Blockheaders nachverfolgen. Die beiden *root* Felder sind jeweils die Wurzel eines *Merkle-Patricia* Baumes, welche es ermöglicht den zugrunde liegenden *trie* zu traversieren. [4] Mit der *transaction-Root* können wir alle im im Block enthaltenen Transaktionskorpora abrufen. Der *trie* zum *receiptRoot* enthält Informationen zu dem Ergebnis der Transaktionen. Ein *receipt* besteht aus dem Zustand R_σ nach der Transaktion, dem kumulierten, verbrauchten Gas nach der Transaktion R_u und den Logs R_l, die zur Laufzeit der Transaktion entstanden sind. Zusätzlich wird noch ein sogenannter *bloom filter* R_b von den Logs angelegt, um eine schnellere Suche zu ermöglichen. [7]

$$R \equiv (R_\sigma, R_u, R_l, R_b)$$

Dieser Bloom Filter ist zusätzlich in dem Feld *logsBloom* abgelegt, wodurch Clients bedeuete schneller relevante Informationen aus dem Block filtern können. [26, S. 5] So schließlich ist auch die Transaktion T_x ein unveränderbarer Teil der Ethereum Blochain geworden.

5 Ausblick

Transaktionen in Ethereum sind ein mächtiges und vielseitiges Werkzeug. Sie sind Teil des Rückgrates des angestrebten Weltcomputers. Allerdings gibt es noch Probleme bei der Leistungsfähigkeit und der Zuverlässigkeit des Netzwerkes, sodass sich ihr wahres Potential möglicherweise noch nicht entfaltet hat.

Literatur

1. Cryptoeconomics In 30 Minutes by Vitalik Buterin (Devcon5)
2. Ethereum Mining Profitability Calculator. https://www.cryptocompare.com/mining/calculator/eth?HashingPower=
3. Ethereum/EIPs. https://github.com/ethereum/EIPs
4. Ethereum/patricia. https://github.com/ethereum/wiki/wiki/Patricia-Tree
5. Ethereum/wiki_pos. https://github.com/ethereum/wiki
6. Frequently Asked Questions — Ethereum Homestead 0.1 documentation. http://ethdocs.org/en/latest/frequently-asked-questions/frequently-asked-questions.html#what-s-the-difference-between-account-and-wallet-contract
7. Logs - How does Ethereum make use of bloom filters? https://ethereum.stackexchange.com/questions/3418/how-does-ethereum-make-use-of-bloom-filters
8. Multisignature - Bitcoin Wiki. https://en.bitcoin.it/wiki/Multisignature
9. Transigere - Translation from Latin into German | PONS. https://en.pons.com/translate/latin-german/transigere
10. Blockchain-etl/ethereum-etl. Blockchain ETL (Apr 2020)
11. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: building smart contracts and DApps. O'Reilly, Sebastopol, CA, first edition edn. (2019), oCLC: ocn967583559

12. Bellare, M., Neven, G.: Identity-Based Multi-signatures from RSA. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Abe, M. (eds.) Topics in Cryptology – CT-RSA 2007, vol. 4377, pp. 145–162. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
13. BitMEX: DDoS attack, 13 March 2020 | BitMEX Blog
14. Davis, M.: Computability and Unsolvability. Courier Corporation (Apr 2013)
15. Ehrlicher, W.: Kompendium der Volkswirtschaftslehre. Vandenhoeck & Ruprecht (1975)
16. Herold, H., Lurz, B., Wohlrab, J., Hopf, M.: Grundlagen Der Informatik. Pearson, third edn. (2017)
17. Iyer, K., Dannen, C.: Cryptoeconomics and Game Theory, pp. 129–141. Apress, Berkeley, CA (2018)
18. M.Spain, M.Foley: OASIcs-Tokeneconomics. Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany (2019)
19. Myerson, R.B.: Game Theory: Analysis of Conflict. Harvard University Press (1997)
20. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System
21. Neemann, J., Claussen, T.: Appendix: Scripts. https://github.com/campfireman/SEM-ethereum-transactions
22. Pierro, G.A., Rocha, H.: The Influence Factors on Ethereum Transaction Fees. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 24–31. IEEE, Montreal, QC, Canada (May 2019). https://doi.org/10.1109/WETSEB.2019.00010
23. Research, B.: Empty Block Data by Mining Pool | BitMEX Blog. https://blog.bitmex.com/empty-block-data-by-mining-pool/
24. Roeder, T.: Asymmetric-Key Cryptography. https://www.cs.cornell.edu/courses/cs5430/2013sp/TL04.asymmetric.l
25. Svanevik, A.: Why All These Empty Ethereum Blocks? https://medium.com/@ASvanevik/why-all-these-empty-ethereum-blocks-666acbbf002 (Oct 2018)
26. Wood, D.G.: ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER p. 32
27. Wood, G.: Ethereum/yellowpaper. 2019-10-20 (Oct 2019)