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

Zusammenfassung. Eine explorative Betrachtung von Transaktionen in Ethereum anhand eines Beispiels. Erläuterung der technischen Spezifikationen und Untersuchung des aktuellen Zustands von Transaktionen.

Schlüsselwörter: Ethereum · Transaktionen · RLP · Gas · Propagation

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 [12]. 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 [19, 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 [20, 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 den oben genannten (ACID) Transaktionen in Datenbanksystemen bei. Der Automat speichert seinen Zustand σ_t in der Blockchain. Eine Transaktion T ist Argument der Zugstandsübergangsfunktion Υ, die von *externen Akteuren* (also auch Maschinen) 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 Konsenses 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 größer, denn Zweck ist nicht nur die Schöpfung, Speicherung und der Austausch eines digitalen Zahlungsmittels [24], sondern eine allgemeine dezentrale Rechenmaschine, ein „Weltcomputer“ [30, 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 Struktur einer Transaktion ist vergleichbar mit der eines Briefes. Es gibt jeweils einen Absender, Empfänger und eine Bezahlung der Zustellungskosten. Bei Transaktionen erfolgt dies nicht in Form einer Briefmarke, sondern durch Gas (s. 3.3). Außerdem kann beides eine „Nutzlast“ [15, S. 108] enthalten. Dabei handelt es sich meistens um einen Etherbetrag und/oder zusätzliche Daten. Genauso wie man in einem Brief Geld und/oder einen Text verschicken kann. Im Folgenden wird die allgemeine Struktur und die technische Umsetzung von Transkation in Ethereum vorgestellt.

2.1 Komponenten einer Transaktion

Transkationen, so auch unsere Transaktion T_x, enthalten laut ihrer offiziellen Definition [30, S. 4] folgende Datenfelder:

nonce: Ein Skalar, welcher gleich der Anzahl vom *External Owned Account* (EOA)[1] 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. 3.3).
gasLimit: Ein Skalar, der die maximale 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 den Betrag an 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 eines *Smart Contracts* enthält.

Im Verlaufe dieser Ausarbeitung werden wir diese Felder für unsere Transaktion T_x füllen und auf die genaue Bedeutung und auf weitere technische Spezifikationen dieser eingehen.

2.2 Typen von Transaktionen

Es gibt genau zwei Typen von Transaktionen, die in der Blockchain dokumentiert werden. Transaktionen, die eine Nachricht von einem EOA zu einem anderen EOA oder Kontrakt überträgt („message calls“ [30, S. 4]) oder Transaktionen, welche einen neuen Kontrakt erzeugen („contract creation“ [30, S. 4]). Mit dem Wort „Nachricht“ ist dabei der Inhalt von den Feldern *value* und *data* gemeint.

[1] Ein Account im Ethereum-Netzwerk, der einem Menschen gehört.

Bei Message-Call-Transaktionen enthält das *to*-Feld die öffentliche Adresse eines EOA oder eines Kontrakts. Unsere Transaktion T_x adressiert einen Kontrak mit der Adresse `0xd76595f64aaf9a79f27cf6831788f7575f0c7f38`. Zusätzlich besteht die Option die Felder *value* und *data* zu füllen.

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“ [15]. Zusätzlich können Ether mitgesendet werden. Diese dienen als Startfinanzierung für den Kontrakt [30, S. 4]. Es sollten jedoch keine Ether an diese Adresse mit einer Message-Call-Transkation gesendet werden, da diese sonst verloren sind und nicht mehr zurückerstattet werden können [15, S. 112].

Ein spezieller Typ von Transaktion ist eine interne Transaktion. Diese treten nur dann auf, wenn ein Kontrakt eine Transkation ausführt [15, S. 40]. Beispielsweise könnte unsere Transaktion T_x dazu dienen Ether von einem Kontraktkonto abzuheben. Wir würden somit eine Message-Call-Transaktion durchführen, die den entsprechenden Funktionsaufruf enthält (s. auf Etherscan). Der Kontrakt sendet uns dann den entsprechenden Etherbetrag zurück. Diese, vom Kontrakt ausgelöste Transaktion, ist eine interne Transaktion und wird nicht in der Blockchain dokumentiert (s. auf Etherscan).

Verteilung der Transaktionen 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* [13] und der API von *infura* lässt sich ein guter Eindruck von dem aktuellen Zustand des *mainnets* von Ethereum gewinnen. Der Datensatz D enthält ein Gesamtvolumen M an 22.748.700 Transaktionen, wovon 2,1% Kontrakterzeugungen sind.

Wir definieren den Gesamtwert einer Transaktion als $V_{\text{total}} = \mathit{gasPrice} \times \mathit{gasLimit} + \mathit{value}$, wobei wir diesen Wert der Empfängeradresse zuschreiben. Ordnet man die Addressen nach V_{total}, macht der kumulierte Wert $\text{SUM}_{top100}(V_{\text{total}})$ der ersten 100 Adressen 2,3% des Gesamtaggregats $\text{SUM}_{all}(V_{\text{total}})$ aus. Alle Adressen der Top 100 sind Kontrakte. Allerdings ist die Anzahl der Transaktion, die an diese Kontrakte addressiert sind, 44,2% des gesamten Transaktionsvolumens M, wobei 15% von M allein an den Kontrakt des Tokens „Tether USD“ addresiert sind [25].

2.3 Serialisierung

Um Daten bzw. Objekte effizient und vor allem konsistent über das P2P-Netzwerk verschicken zu können, benötigt man eine Art der Serialisierung. Ethereum greift dabei auf das „Recursive Length Prefix“ (RLP) Verfahren zurück. Dies ist ein simples Kodierungsverfahren und ermöglicht eine „byte-perfekte“ Konsistenz [6].

Auch die Daten unserer Transaktion T_x werden mit Hilfe von RLP serialisiert und deserialisiert. Die RLP-Funktion erhält ein sogenanntes „Item“ als Parameter. Ein Item ist dabei ein String (z. B. ein Byte-Array) oder eine (verschachtelte)

Liste von Items. Das Verfahren setzt nun Präfixe abhängig von der Länge des Items. Je nach Länge gelten für die Präfixe verschiedene Regeln [8]:

1. Item ist ein String (z. B. 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“:

     `[0x88, 'E', 't', 'h', 'e', 'r', 'e', 'u', 'm']`

     bzw inkl. ASCII-Kodierung

     `[0x88, 0x45, 0x74, 0x65, 0x68, 0x72, 0x65, 0x75, 0x6d]`[2]
   - 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 Strings 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. Insgesamt erhalten wir $\texttt{0xb7} + 2 = \texttt{0xb9}$ als Präfix.
2. Item ist eine (verschachtelte) Liste von Items:
   - Ist die Gesamtlänge aller 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 `["Ether", "Wei"]`:

     `[0xca, 0x85, 'E', 't', 'h', 'e', 'r', 0x83, 'W', 'e', 'i']`

     bzw. inkl. 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 und 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 Präfixes in allen Fällen länger als 255 ist und somit nicht in einem Byte dargestellt werden kann [30, S.18,19].

[2] Aufgrund besserer Lesbarkeit werden die RLP-Repräsentationen in Listenform dargestellt. In der Realität ist dies jedoch eine einfache Bytefolge.

3 Aufbau einer Transaktion

3.1 Nonce

Das Konzept der Nonce wurde nicht von Ethereum eingeführt, sondern kommt aus dem Bereich der Kryptografie. Eine Nonce ist dort laut Definition [8] eine willkürliche Nummer, die nur einmal in einer kryptographischen Kommunikation verwendet wird.
In Ethereum-Transaktionen ist die Nonce eine Zahl, welche bei der Accounterstellung den Wert Null hat und bei jeder erfolgreichen Transaktion[3] um eins inkrementiert wird. Dieser Wert wird dabei nicht explizit im Account gespeichert. Stattdessen wird die Anzahl der erfolgreichen Transkationen dynamisch gespeichert [15, S.101].

Mit der Nonce werden sogenannte „Replay-Attacken“ verhindert. Die Blockchain kann von jedem eingesehen werden und somit lässt sich jede Transaktion aus der Blockchain kopieren und theoretisch unbegrenzt erneut ausführen. Um dem entgegenzuwirken, wird die Nonce in der Transkation hinterlegt. Wenn diese nun erneut ausgeführt werden soll, wird diese von der Blockchain abgelehnt, da die Nonce der Transaktion nicht mehr mit der Nonce des originalen Transaktionsabsenders übereinstimmt. Daraus folgt, dass man seine eigenen Transaktionen auch nicht wiederholen kann.

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 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 Accounts. Stimmen beide überein, wird die Transkation sofort verarbeitet. Ist die Nonce der Transaktion größer als die erwartet, landet die Transkation im *Mempool*, in dem 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. Die Transaktion bleibt solange im Pool, bis die Transaktionen mit den Nonce 2, 3 und 4 im Netzwerk registriert wurden. Somit kann man eine Priorisierung von Transaktionen ermöglichen. In dem man weniger wichtigere Transkationen bzw. Transaktionen, die von der erfolgreichen Ausführung vorheriger Transaktionen abhängen, mit einer höheren Nonce absendet.

3.2 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

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

dabei auch Zahlung bzw. *payment* genannt. Das *data*-Feld ist ein Aufruf bzw. *invocation* [15, S.108]. Eine Zahlung zwischen zwei EOAs ist dabei eine einfache Zustandsänderung der EVM und Übertragung des Etherbetrags in *Wei* auf dem in *to*-Feld hinterlegten Account. Enthält diese Transkation Daten im *data*-Feld, so werden diese von der Blockchain ignoriert [30, S.10]. In einer Wallet werden diese meisteins nur angezeigt (s. 1).

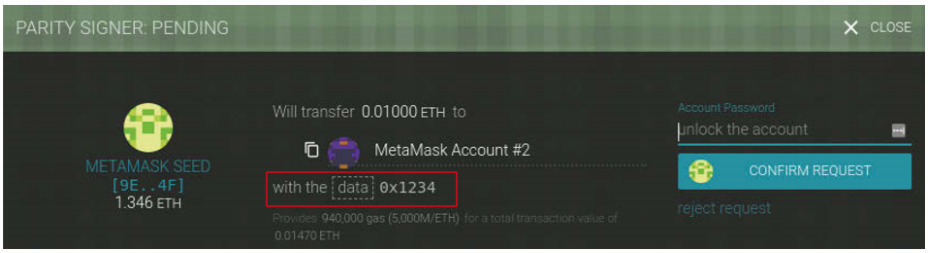

Abb. 1. Beispieltranskation an EOA mit gefülltem *data*-Feld [15, S.109]

Wie bereits in 2.2 erwähnt, wird der Inhalt des *data*-Feld erst von Interesse, wenn wir einen Kontrakt ansprechen. Beispielsweise soll unsere Transkation die Funktion

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

des Kontrakts (s. auf Etherscan) aufrufen. Diese Funktion fügt dem kontraktinternen Konto, dem im *value*-Feld übergeben Wert hinzu. Dabei muss als Parameter der Einzahlungsgrund genannt werden.

Damit jede Transaktion eine Funktion nach dem gleichen Schema aufruft, wird in der *Contract Application Binary Interface* (ABI) Spezifikation von Solidity definiert, wie ein Funktionsaufruf im *data*-Feld hinterlegt werden muss [1]. Der Funktionsaufruf 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 des Funktionsprototypen. Keccak-256 ist das am meisten verwendete Hashverfahren in Ethereum [30, S. 3]. 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 Keccak-256-Hash entspricht:

`0xa26e11860cdb80ecca46e4f433c3c9533f6d37cdf0f6eb16343556cfdbcf47ec`

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 Parameters startet, konkateniert 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 \
  45696e7a61686c756e67000000000000000000000000000000000000000000000
```

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

```
                                0xa26e1186 \
0000000000000000000000000000000000000000000000000000000000000020 \
000000000000000000000000000000000000000000000000000000000000000a \
 45696e7a61686c756e67000000000000000000000000000000000000000000000
```

Man kann jedoch über das *data*-Feld keinen Ether an den Kontrakt übergeben. Dies ist ausschließlich über das *value*-Feld möglich. Damit der Kontrakt diesen Ether annimmt, ist unsere Funktion mit dem Schlüsselwort `payable` deklariert sein. Akzeptiert die aufgerufene Funktion keinen Ether, so wird die sogenannte Fallback-Funktion aufgerufen, die den übergebenen Etherbetrag auf das Konto des Kontrakts gutschreibt. Ist auch diese nicht definiert, wird eine Exception geworfen und die Transaktion abgebrochen [1].

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

3.3 Gas

Gas ist ein zentraler konzeptioneller Lösungsansatz im Rahmen von Ethereum. Da Ethereum turing-vollständig ist [30, 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 [18, 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 Ressourcenverbrauch in der EVM bemisst [22, S. 9:3], wobei für jeden Opcode die Kosten in Gas spezifiziert werden [30, S. 25 ff.]. So werden neben Kosten für Rechenaufwand auch Kosten für die Nutzung von persistentem Speicher miteinbezogen. Es gilt sogar die 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. Da Instruktionsblöcke aber zum Beispiel vom Zustand des Kontraktes abhängen, kann dies nur grob vorgenommen werden. Eine robuste Programmierung von *Smart Contracts* ist essenziell. Ein erster Anhaltspunkt dafür sind zunächst die intrinsischen Kosten einer Transaktion. Das ist der Overhead der allein durch die Transaktion und deren Inhalt besteht. Diese intrinsischen Kosten g_0 lassen sich mit auf Basis folgender Grundlage berechnen.

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

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 *gas* 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 mit $G_{\text{txdatazero}} \times 84$ und $G_{\text{txdatanonzero}} \times 12$ intrinsische Kosten für die Daten: 3252 *gas*. Hierzu sind dann noch die Kosten $G_{transaction}$ zu addieren [25].

Preis und Latenz Gas kann bewusst nur mit Ether erworben werden, da die Gas-Preise möglichst unabhängig von den Preisschwankungen sein sollen. Der *gasPrice* kann freigesetzt 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. Danach empfiehlt sich für T_x ein *gasPrice* von 9 GWei im *mainnet*. Am 20.04.2020 akzeptieren ungefähr 84% der letzten 200 Blöcke diesen Preis, sodass sich für unsere Transaktion T_x Kosten in Höhe von 3252 ×9Gwei = 218268GWei für die intrinsischen Kosten 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 Veröffentlichung 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 [26, S. 30 f.]. Da das *gasLimit* eines Blockes nach erfolgreichem Schürfen durch den Miner um maximal $\frac{P(H)_{H1}}{1024}$ des alten Limits $P(H)_{H1}$ erhöht

oder verringert werden darf, limitiert dies die Fähigkeit viele Transaktionen in einem kurzen Zeitintervall zu verarbeiten oder dynamisch auf eine höhere Last zu reagieren. Gleichzeitig verhindert dies eine Zentralisierung der Rechenleistung auf wenige, große Miner durch zu große und rechenintensive Blöcke.

Zieht man den Datensatz D zu Rate, ergibt sich aktuell folgender durchschnittlicher Durchsatz T_{max} pro Block [25]:

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

Übersteigt die Zahl der Transaktionen diese Durchsatzfähigkeit des Netzwerkes signifikant, wie beispielsweise bei einem *ICO (initial coin offering)*, führt dies zu einer Eskalation der Transaktionskosten, um zum Beispiel möglichst schnellen Zugriff auf die Wertanlagen zu erhalten [22, S. 9:6 f.]. Im betrachteten Zeitraum war dies zum Beispiel am 13.03.2020 der Fall (s. 2). Zu diesem 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 [17].

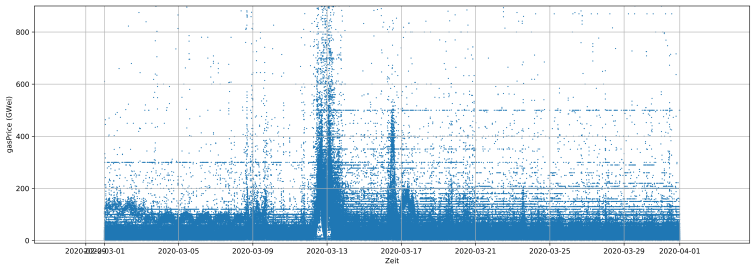

Abb. 2. gasPrice nach Tag im Monat März [25]

Bei Betrachtung der Verteilung der Anzahl Transaktionen pro Block 4 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 Blöcken, die fast keine oder sehr wenige Transaktionen enthalten und gleichzeitig nur einen Bruchteil des *blockGasLimit* ausschöpfen, wobei sogar 3,4% der Blöcke komplett leer sind [25]. 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. Deswegen wird beim sogenannten *SPV-Minig* sogar komplett auf die Verifizierung verzichtet [29].Hier erhöhen mehr Transaktionen die Gefahr eines *double-spend* und die Ungültigkeit des Blockes. Außerdem verbrauchen mehr Transaktionen mehr Speicher, ergo desto mehr Bandbreite benötigt die Veröffentlichung des Blocks [27].

Alle Vermutungen deuten in die gleiche Richtung: Je mehr Zeit verbraucht wird, desto höher das Risiko, dass keine Belohnung ausgezahlt wird, da ein anderer Miner seine Lösung schneller veröffentlicht. Also gilt es möglichst das Optimum zwischen verbrauchter Zeit und dem Profit zu finden.

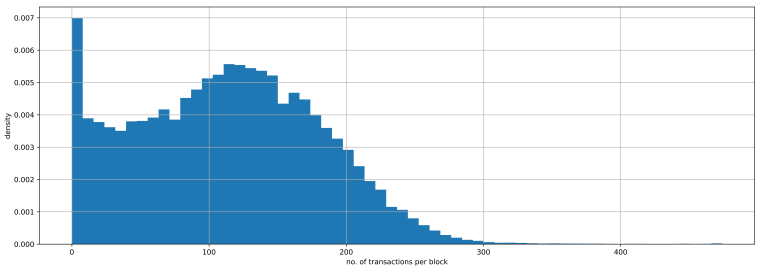

Abb. 3. Verteilung der Zahl an Transaktionen pro Block (60 konstante Klassen) [25]

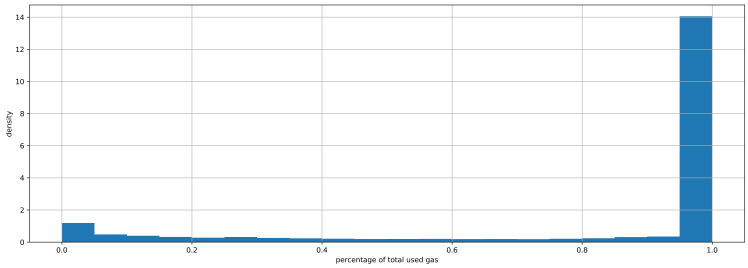

Abb. 4. Verteilung des genutzten Gases pro Block in Prozent (20 konstante Klassen) [25]

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 ausgegangen werden, dass alle altruistisch motiviert sind und aus Idealismus ihren Beitrag leisten. Ethereum geht aber von einer anderen Prämisse aus. Das Feld der *Cryptoeconomics* beschreibt wie ökonomische Anreize genutzt werden, um die Funktion von Kryptowährungen zu garantieren. Die Modellierung von Ethereum und seinem Anreizsystem basiert auf spieletheoretischen Ansätzen [2]. In der Spieletheorie wird davon ausgegangen, dass Entscheidungsträger rational sind und sie die Nützlichkeit *(utility payoff)* ihrer Entscheidungen, unter Berücksichtigung ihres Zustandes t und der Menge der möglichen Belohnungen X, maximieren wollen. Unter dieser Annahme muss eine Kryptowährung also Anreize, also Belohnungen $x \in X$, schaffen, die für den Fortbestand des Systems zuträglich sind und gleichzeitig die *utility payoff* Nützlichkeitsfunktion $u(x, t)$ möglichst maximieren [23, 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 Berücksichtigung der aktuellen *block rewards*, *block difficulty*, etc. sehr gut errechnen [3]. Nun kann sich durch Erfahrungswerte des rationalen Entscheidungsträgers herausstellen, dass die Nützlichkeit in manchen Situationen höher liegt, wenn keine Transaktionen in Blöcken inkludiert werden. Als Resultat sind rein wirtschaftlich motivierte Akteure 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, da jeder zusätzliche, ungetestete Mechanismus der Währung eine größere Angriffsfläche bietet [7].

3.4 Signatur

Die Signatur einer Transaktion belegt den Besitz eines Schlüssels, der aktuell die *Authentizität* 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 [28].

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 Keccak-256-Hash von K kann daraus die Ethereum Adresse 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*, *gas-*

Limit, *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 [4] 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 [21, 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, ein weiterer Angriffsvektor für eine *Replay-Attacke*.

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

$$sig = F_{sig}(F_{Keccak256}(m), k)$$

ECDSA Die resultierende Signatur *sig* besteht aus den Komponenten r, s und dem Signaturpräfixwert v. Diese drei Komponenten sind Teil einer speziellen Form des *Digital Signature Algorithm*, die Elliptische-Kurven-Kryptografie 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* basiert die *Trapdoor-Funktion* auf dem "Problem des diskreten Logarithmus in elliptischen Kurven", für das noch kein besserer Ansatz als die naive Variante gefunden wurde. Entsprechend sind deutlich kleinere Schlüsselgrößen möglich [14].

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. Der 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 Transaktionen 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: [15, S. 114 ff.]

```
v: 0x26
r: 0xdade772f31d20b4ed1c7f63ae035c0cc83fd7b786ca9339eb01763138877a6d4
s: 0x13e16f7a55d261e504e27ea4fecc174a1a46c87d804b9a3917aebde665c1ddb1
```

Multisignaturen Eine spezielle Form der Signatur sind Multisignaturen. Bei einem *multi-signature* Schema muss eine Nachricht t von n aus m Entitäten unterzeichnet werden, wobei $m \geq n > 1$ [16, 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 [11], bedarf es bei Ethereum die Verwendung eines Kontrakts [9].

4 Transaktionsabwicklung

Wir haben nun alle notwendigen Konzepte und Inhalte einer Ethereumtransaktion kennengelernt und können alle Felder unserer Transaktion füllen. Für T_x gilt jetzt somit:

```
T = {
  nonce: 0x6,
  gasPrice: 0x3B9ACA00,
  gasLimit: 0x1117A,
  to: 0xd76595f64aaf9a79f27cf6831788f7575f0c7f38,
  data: 0xa26e11860...a45696e7a61686c756e670...0,
  v: 0x26
  r: 0xdade772f...8877a6d4,
  s: 0x13e16f7a...65c1ddb1
}
```

Nun werden alle Daten (die Feldnamen sind nicht Teil der Transaktion [15, S. 100]) serialisiert.

4.1 Propagation

Bevor unsere signierte Transaktion über das Ethereum-Netzwerk verbreitet wird, überprüft der lokale Node, 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 [15, S. 123]. Jeder dieser Nodes erhält die Transkation und validiert diese. Wenn dies erfolgreich ist, sendet der jeweilige Node die Transaktion an seine Nachbarn weiter [15, S. 123]. So wird erreicht, dass die Transkation sehr schnell bei jedem Node im Ethereum-Netzwerk angekommen ist. Die Transaktion erreicht somit auch sogenannte „Miner Nodes“. Diese Nodes speichern unsere Transaktion in ihrem *Mempool*. Abhängig von unserer Platzierung in diesem Pool, wird die Transaktion nach einer Zeit in einen Block aufgenommen. Sobald der Block geschürft wurde, also der *Proof of Work* gefunden wurde, findet eine Zustandsänderung des Zustandsautomat statt. Das heißt, die Funktion des Kontrakts wird von der EVM ausgeführt und der Ether von dem Senderaccount abgebucht und dem Konto des Kontrakts gutgeschrieben. Schließlich kann man unsere Transaktion in der Blockchain wiederfinden: Unsere Transaktion auf Etherscan

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 [5]. Mit der *transactionRoot* können wir alle 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 [10].

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

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. Durch einen neuen Konsensmechanismus, namentlich *proof of stake*, könnte der Durchsatz an Transaktionen drastisch steigen. Auch eine Unterstützung von homomorphischer Verschlüsselung der Transaktionsdaten und *zero-knowledge-proofs* könnten Teil des offiziellen Protokolls werden und eine neue Klasse von Anwendungen ermöglichen.

Literatur

1. Contract ABI Specification — Solidity 0.6.6 documentation. https://solidity.readthedocs.io/en/v0.6.6/abi-spec.html
2. Cryptoeconomics In 30 Minutes by Vitalik Buterin (Devcon5)
3. Ethereum Mining Profitability Calculator. https://www.cryptocompare.com/mining/calculator
4. Ethereum/EIPs. https://github.com/ethereum/EIPs
5. Ethereum/patricia. https://github.com/ethereum/wiki/wiki/Patricia-Tree
6. Ethereum/wiki. https://github.com/ethereum/wiki
7. Ethereum/wiki_pos. https://github.com/ethereum/wiki
8. Ethereum/wiki/RLP. https://github.com/ethereum/wiki
9. 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
10. Logs - How does Ethereum make use of bloom filters? https://ethereum.stackexchange.com/questions/3418/how-does-ethereum-make-use-of-bloom-filters
11. Multisignature - Bitcoin Wiki. https://en.bitcoin.it/wiki/Multisignature
12. Transigere - Translation from Latin into German | PONS. https://en.pons.com/translate/latin-german/transigere
13. Blockchain-etl/ethereum-etl. Blockchain ETL (Apr 2020)
14. Elliptic Curve Digital Signature Algorithm. Wikipedia (Mar 2020)
15. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: building smart contracts and DApps. O'Reilly, Sebastopol, CA, first edition edn. (2019)

16. 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)
17. BitMEX: DDoS attack, 13 March 2020 | BitMEX Blog
18. Davis, M.: Computability and Unsolvability. Courier Corporation (Apr 2013)
19. Ehrlicher, W.: Kompendium der Volkswirtschaftslehre. Vandenhoeck & Ruprecht (1975)
20. Herold, H., Lurz, B., Wohlrab, J., Hopf, M.: Grundlagen Der Informatik. Pearson, third edn. (2017)
21. Iyer, K., Dannen, C.: Cryptoeconomics and Game Theory, pp. 129–141. Apress, Berkeley, CA (2018)
22. M.Spain, M.Foley: OASIcs-Tokeneconomics. Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany (2019)
23. Myerson, R.B.: Game Theory: Analysis of Conflict. Harvard University Press (1997)
24. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System
25. Neemann, J., Claussen, T.: Appendix: Scripts. https://github.com/campfireman/SEM-ethereum-transactions
26. 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
27. Research, B.: Empty Block Data by Mining Pool | BitMEX Blog. https://blog.bitmex.com/empty-block-data-by-mining-pool/
28. Roeder, T.: Cryptography: Asymmetric-Key. https://www.cs.cornell.edu/courses/cs5430/2013sp/TL04.asymmetric.html
29. Svanevik, A.: Why All These Empty Ethereum Blocks? https://medium.com/@ASvanevik/why-all-these-empty-ethereum-blocks-666acbbf002 (Oct 2018)
30. Wood, G.: Ethereum/yellowpaper. 2019-10-20 (Oct 2019)