**HOCHSCHULE**
**HANNOVER**
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
*Fakultät IV*
*Wirtschaft und*
*Informatik*

# Ethereum

*Transaktionen*

Ture Claußen und Jannes Neemann, 27.05.2020

# Inhaltsverzeichnis



# Inhaltsverzeichnis

# Einleitung

- lat.: „transigere” bedeutet soviel wie „duchführen”, „vollführen” oder „abmachen”
- Ethereum ist ein „transaktionsbasierter Automat“ (*transaction-based state machine*)

# Inhaltsverzeichnis



# Komponenten einer Transaktion

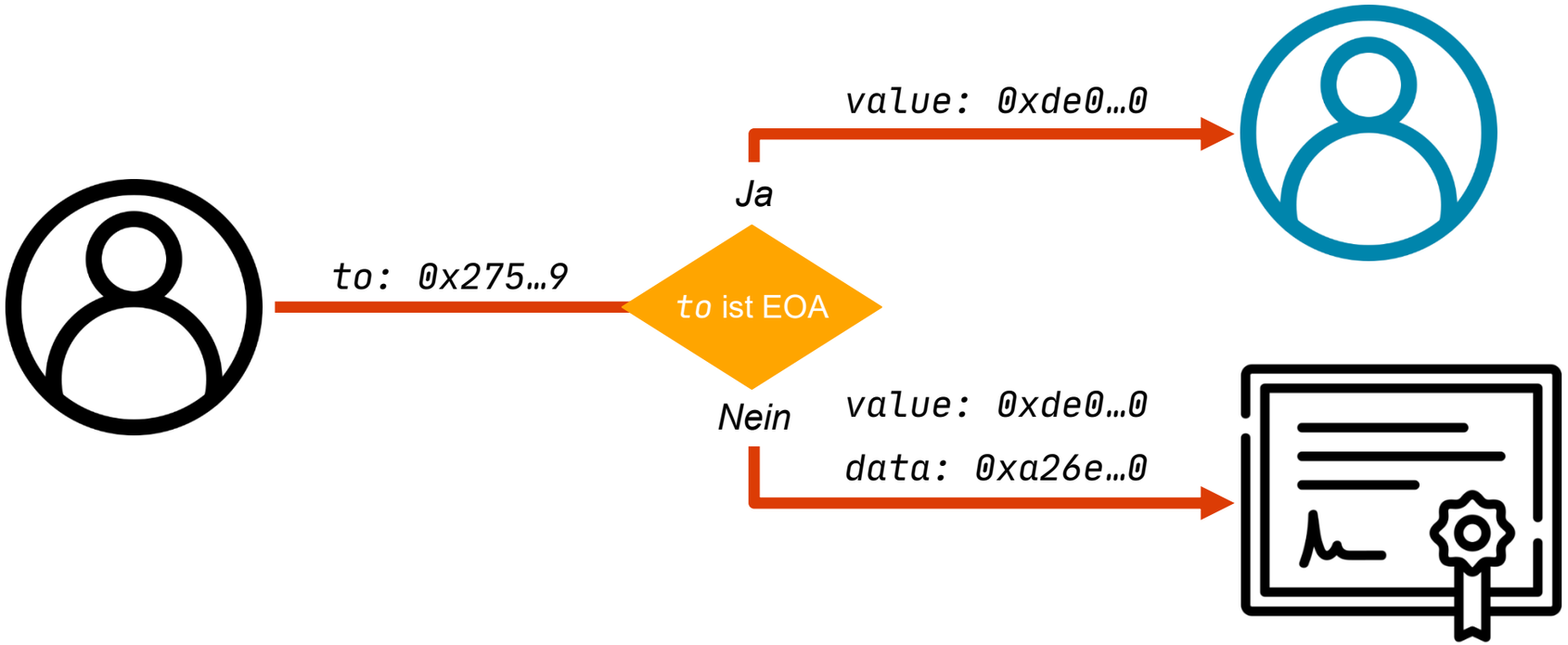
$T_x$ = {

| | |
|---|---|
| `nonce:` | Skalar = Anzahl vom EOA durchgeführte Transaktionen |
| `gasPrice:` | Skalar = Betrag an Wei pro Gas |
| `gasLimit:` | Skalar = Maximal Betrag an verbrauchbarem Gas |
| `To:` | Skalar = 160-Bit Adresse des Empfängers |
| `value:` | Skalar = Betrag an Wei, den der Empfänger erhält |
| `v,r,s:` | Skalare = Komponenten der ECSDA-Signatur |
| `init:` | Byte-Array = kompilierter Sourcecode des Kontrakts |
| `data:` | Byte-Array = Funktionsaufruf eines Kontrakts |

}

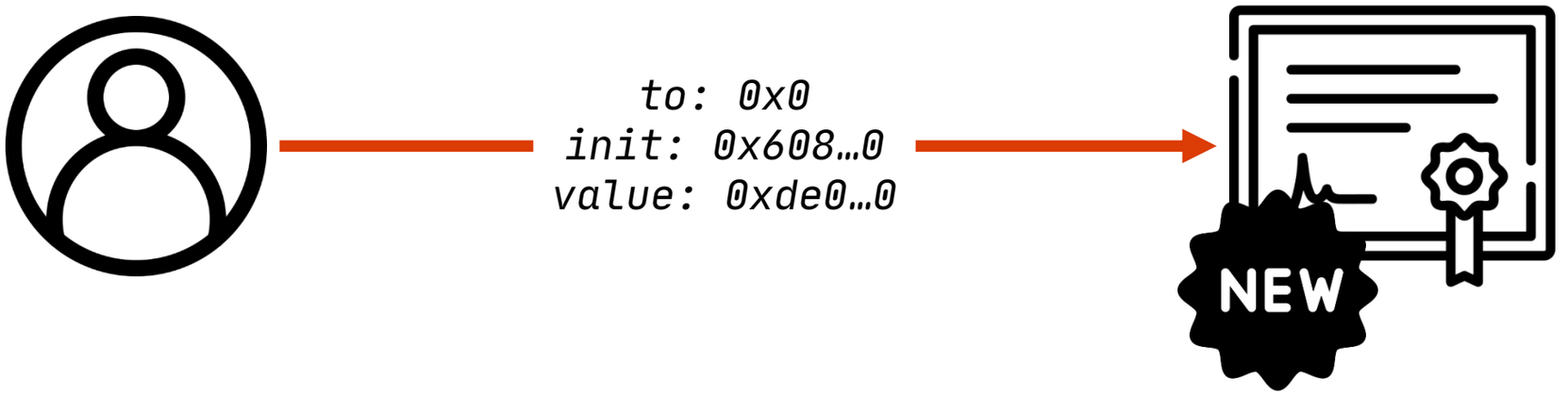
# Typen von Transkationen

1. Message-Call-Transaktionen



# Typen von Transkationen

2. Contract-Creation-Transaktionen



# Verteilung

# Serialisierung

*Recursive Length Prefix (RLP)*

- Ausschließlich zur Serialisierung von Struktur
- Funktionsparameter ist ein Item
- Item entweder String (z.B. Byte-Array) oder Array von Items
- Präfix wird abhängig von Länge des Items gesetzt
- Verschiedene Regeln je nach Item

*Ethereum* = `['E','t','h','e','r','e','u','m']` < 56 Bytes ⇒ Präfix: `0x80` + 8 Bytes Zeichenlänge

⇓

`[0x88, 0x45, 0x74, 0x65, 0x68, 0x72, 0x65, 0x75, 0x6d]`

*[„Ether“, „Wei“]* = Gesamtlänge < 55 Bytes ⇒ Präfix `0xc0` + RLP Repräsentation der einzelnen Items

⇓

`[0xca, 0x85, 0x45, 0x74, 0x68, 0x65, 0x72, 0x83, 0x57, 0x65, 0x69]`

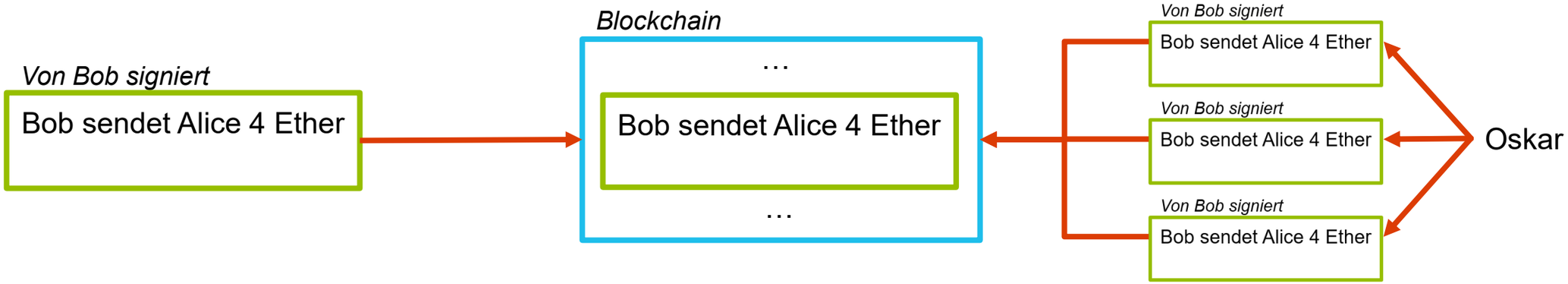
# Inhaltsverzeichnis

# Nonce

- Konzept aus der Krythographie:

  “ a nonce is an arbitrary number that can be used just once in a cryptographic communication”[1]

- Inkrementierende Zahl ermöglicht Einmaligkeit einer Transaktion
- Schutz gegen “Replay-Angriffe”



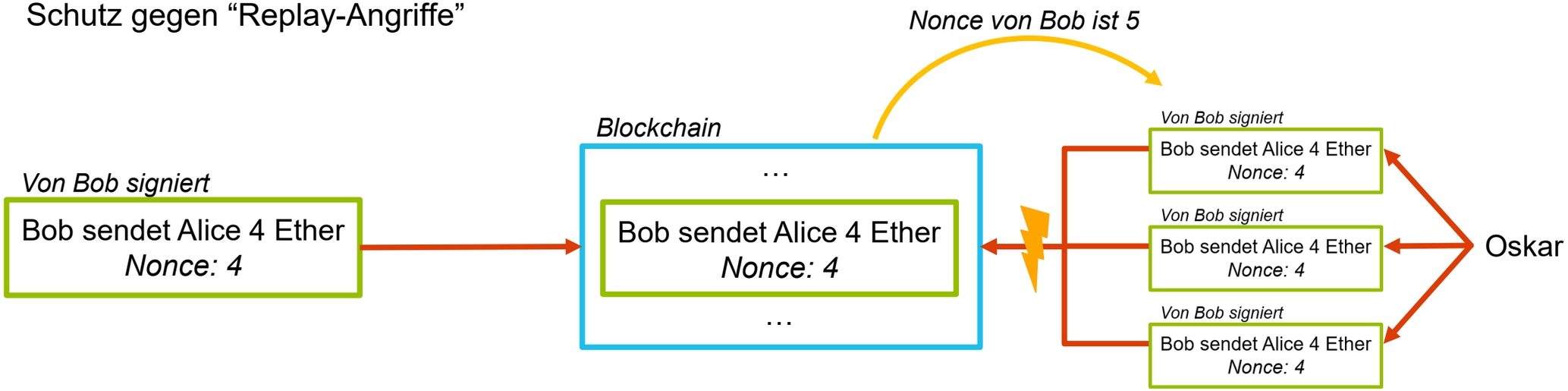

[1]Quelle: Wikipedia, Cryptographic nonce
https://en.wikipedia.org/wiki/Cryptographic_nonce

# Nonce

- Konzept aus der Krythographie:

  “ a nonce is an arbitrary number that can be used just once in a cryptographic communication”[1]

- Inkrementierende Zahl ermöglicht Einmaligkeit einer Transaktion
- Schutz gegen “Replay-Angriffe”




[1]Quelle: Wikipedia, Cryptographic nonce
https://en.wikipedia.org/wiki/Cryptographic_nonce

# Value und Data

*Das data-Feld*

- $T_x$ soll folgende Funktion aufrufen:

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

# Value und Data

*Das data-Feld*

- $T_x$ soll folgende Funktion aufrufen:

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

- Funktion besitzt das Schlüsselwort `payable`, damit sie Ether entgegen nehmen kann

# Value und Data

*Der Funktionsaufruf*

- $T_x$ soll folgende Funktion aufrufen:

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

# Value und Data

*ABI-Spezifikation*

- Contract Application Binary Interface (ABI)
- Definiert wie mit einem Kontrakt über Transaktionen interargiert werden muss
- ABI konformer Funktionsaufruf besteht aus zwei Komponenten:
  1. Funktionsselektor
  2. Funktionsargumente

# Value und Data

*ABI-Spezifikation*

1. Funktionsselektor
   - Welche Funktion aufgerufen werden soll
   - Ersten vier Bytes des Keccak-256-Hash des Funktionsprototypen

# Value und Data

*ABI-Spezifikation*

1. Funktionsselektor
   - Welche Funktion aufgerufen warden soll
   - Ersten vier Bytes des Keccak-256-Hash des Funktionsprototypen

*Funktionsprototyp*

```
deposit
```

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

# Value und Data

*ABI-Spezifikation*

1. Funktionsselektor
   - Welche Funktion aufgerufen warden soll
   - Ersten vier Bytes des Keccak-256-Hash des Funktionsprototypen

*Funktionsprototyp*

```
deposit(string)
```

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

# Value und Data

*ABI-Spezifikation*

1. Funktionsselektor
   - Welche Funktion aufgerufen warden soll
   - Ersten vier Bytes des Keccak-256-Hash des Funktionsprototypen

*Funktionsprototyp*

```
deposit(string)
```

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

```
Keccak-256(deposit(string)) = 0xa26e118600cdb80ecca46e4f433c3c9533f6d37cdf0f6eb16343556cfdbcf47ec
```

*Funktionsselektor*

# Value und Data

*ABI-Spezifikation*

2. Funktionsargumente
   - Angabe eines Offsets, der angibt wann das Argument im Hash folgt
   - Konkateniert mit der Länge des Arguments
   - Konkateniert mit dem Argument („Einzahlung“)
   - Jeweils auffüllen mit Paddingsbytes auf 32 Byte

   Offset: 32 Zeichen (`0x20`)

   Länge des Arguments: 10 (`0xa`)

   Argument: “Einzahlung” (`0x45696e7a61686C756e67`)

   Resultat:
```
0x0000000000000000000000000000000000000000000000000000000000000020 \
  000000000000000000000000000000000000000000000000000000000000000a \
  45696e7a61686c756e67000000000000000000000000000000000000000000000
```

# Value und Data

*ABI-Spezifikation*

- Konkatenieren von Funktionsselektor und Funktionsargument ergibt data-Feld Inhalt:

  $T_x$ = {

  `data: 0xa26e11860…020…0a45696e7a61686C756e670…0`

  …

  }

- Es soll 1 Ether dem Kontraktkonto hinzugefügt werden

  1 Ether ≙ 10000000000000000 Wei ≙ `0xde0b6b3a7640000` Wei

  $T_x$ = {

  `data: 0xa26e11860…020…0a45696e7a61686C756e670…0`

  `value: 0xde0b6b3a7640000`

  …

  }

# Gas

*Definition*

- Konzeptioneller Lösungsansatz für das Halteproblem
- Bemisst den Ressourcenverbrauch des Weltcomputers
- Kosten einer Transaktion: *gasPrice* × *gasLimit bzw. gasPrice* × *gasUsed*

# Gas

*Intrinsische Kosten* $g_0$

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

# Gas

## *Intrinsische Kosten $T_x$*

- $T_x$ mit $G_{txdatazero} \times 4$ und $G_{txdatanonzero} \times 68$ → intrinsische Kosten:

  3524 gas + 21000 gas

- Abschätzung: Wie viel Rechenleistung wird zusätzlich gebraucht?

# Gas

*gasPrice von $T_x$*

- Am 20.04.2020 aktzeptieren ungefähr 84% der letzten 200 Blöcke den Preis von 9GWei

# Gas

*Preis und Latenz*

- Korrelation zwischen gasPrice und Latenz?
- Eskalation von Transaktionskosten

# Gas

*Durchsatzfähigkeit*

$$T_x = \frac{blockGasLimit}{transactionMedianGas} = \frac{9817880}{80000} = 122{,}72$$

# Gas

*blockGasLimit*

- um maximal $\frac{P(H)_{H1}}{1024}$ des alten Limits $P(H)_{H1}$ erhöht oder verringert warden darf

# Gas

*Entwicklung gasPrice*

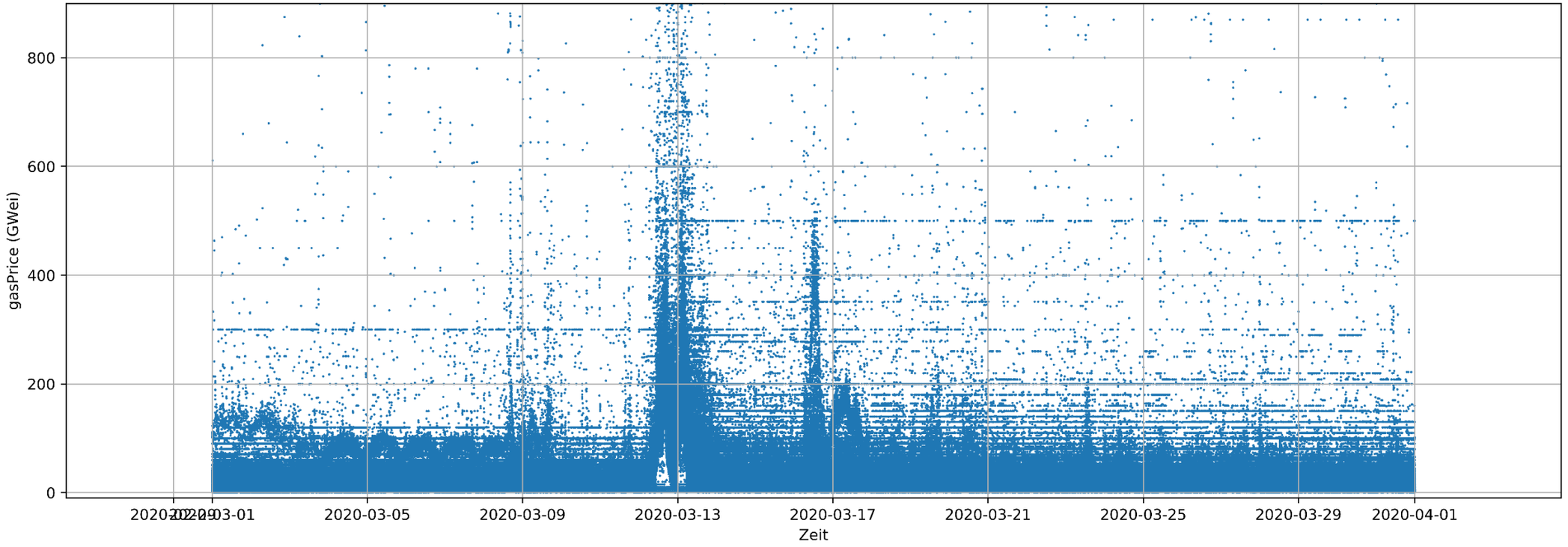



Abbildung: gasPrice nach Tag im Monat März

# Gas

*Preis und Latenz*

- Hohe Auslastung in kleinen Zeitintervallen problematisch
- ICOs und DOS Angriffe verringern Zuverlässigkeit

# Gas

*Preis und Latenz*

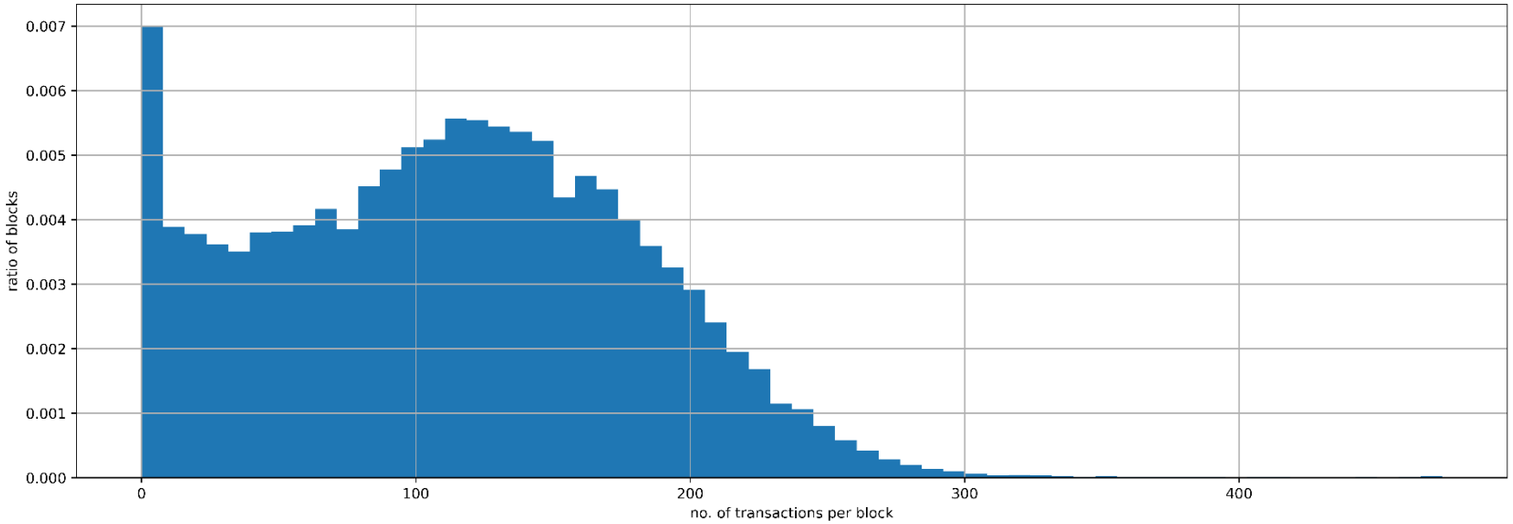



Abbildung: Verteilung der Zahl an Transaktionen pro Block (60 konstante Klassen)

- Leere Blöcke lassen sich schneller Veröffentlichen
- Wirtschaftliche Interessen gehen vor

# Gas

*Anreiz und Spieltheorie*

- Warum sollte ich Ressourcen für das System zur Verfügung stellen?

  → Cryptoeconomics

- Formalisierung des menschlichen Verhaltens durch Spieletheorie
- *utility payoff* Nützlichkeitsfunktion $u(x, t)$ möglichst maximieren

# Signatur
*Bedeutung*

- belegt den Besitz eines Schlüssels, der aktuell die Authenzität und die Integrität der Nachricht beweisen

# Signatur
*Funktion*

$$v, r, s = F_{sig}(F_{keccak256}(m), k)$$

- Serialisierte Form aller Datenfelder + ChainID

# Signatur

*Signatur von $T_x$*

$T_x$ = {

```
    nonce:     0xa
    gasPrice:  0x218711a00
    gasLimit:  0x30000
    v:         26
    r:         0xdade772f31d20b4ed1c7f63ae035c0cc83fd7b786ca9339eb01763138877a6d4
    s:         0x13e16f7a55d261e504e27ea4fecc174a1a46c87d804b9a3917aebde665c1ddb1
    to:        0xb0920c523d582040f2bcb1bd7fb1c7c1ecebdb34
    value:     0xde0b6b3a7640000
    data:      0xa26e11860…020…0a45696e7a61686C756e670…0
```

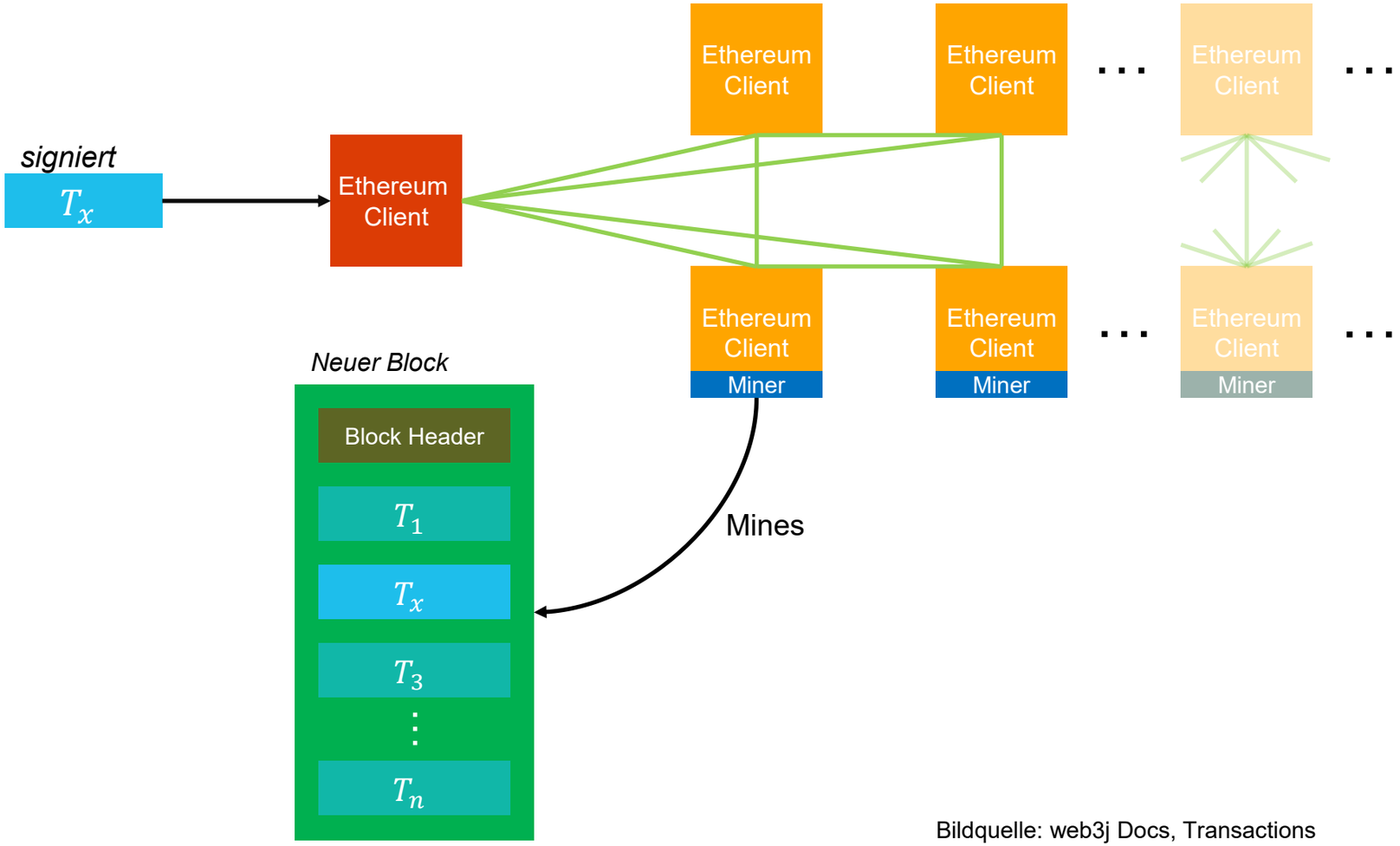
}

# Inhaltsverzeichnis



# Transaktionabwicklung

*Propagation*




Bildquelle: web3j Docs, Transactions
https://web3j.readthedocs.io/en/latest/_images/web3j_transaction.png

# Transaktionabwicklung

*Speicherung*

- Inkludierung im Block
- Erstellung eines Receipts: *receipt* besteht aus dem Zustand $R_\sigma$ nach der Transaktion, dem kumulierten, verbrauchten Gas nach der Transaktion $R_u$ und den Logs $R_l$
- *bloom filter* $R_b$ von den Logs
- Nach Konsens über Block unveränderlicher Teil der Blockchain

# Inhaltsverzeichnis



# Ausblick

- Homorphische Verschlüsselung und Zero-knowledge-proofs

  → Ethereum 2.0

# Vielen Dank für Eure Aufmerksamkeit!