Blockchain Engineering

Smart Contracts & Bitcoin-Script

Dr. Lars Brünjes

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

MODULARES INNOVATIVES
NETZWERK FÜR DURCHLÄSSIGKEIT

AUFSTIEG DURCH
BILDUNG
OFFENE HOCHSCHULEN

17. Oktober 2019

- Smart Contracts (intelligente Verträge) sind (mehr- oder weniger komplexe) Verträge “auf der Blockchain”.
- Während traditionelle Verträge vom Gerichtssystem des Gerichtsstandes des Vertrages durchgesetzt werden, werden Smart Contracts automatisch durchgesetzt (wie gewöhnliche Überweisungen einer Kryptowährung auch).
- Traditionelle Verträge bieten oft Interpretationsspielraum, und ihre Auslegung hängt vom bearbeitenden Richter bzw. Gericht ab. Smart Contracts sind mathematisch eindeutig und absolut präzise.
- Ob das ein Vor- oder Nachteil ist, kann man diskutieren. Zwar sind Gerichte wie alle menschlichen Einrichtungen fehlbar, sie haben aber auch die Möglichkeit, nicht am “Wortlaut” eines Vertrages zu kleben, sondern stattdessen die *Absicht* eines Vertrages mit “gesundem Menschenverstand” zu beurteilen.
- So gesehen sind Smart Contracts “purer Wortlaut”: Die Absicht des Verfassers eines Smart Contracts ist unerheblich, nur der Code zählt.

- Bisher haben wir als Adresse (Absender und Empfänger von Transaktionen) nur Hashs von öffentlichen Schlüsseln betrachtet.
- Selbst bei Bitcoin, ganz zu schweigen von Ethereum, ist die Realität komplizierter.
- Anstelle von statischen Adressen verwendet Bitcoin kleine Programme (Scripts).
- Beim Verifizieren des Inputs einer Transaktion wird der Input-Skript mit dem Output-Skript kombiniert und ausgeführt, und das Ergebnis des Programms entscheidet, ob die ausgebende Transaktion das Recht hat, den entsprechenden Output auszugeben.
- Die Details hängen von der Kryptowährung ab, aber das Prinzip bleibt dasselbe: Programme entscheiden, unter welchen Bedingungen Geld ausgegeben werden darf.

- Die Programme, die entscheiden, unter welchen Bedingungen eine Kryptowährung ausgegeben werden darf, müssen auf allen Knoten laufen, wenn eine Transaktion verifiziert wird (also insbesondere für alle Transaktionen in jedem neuen Block).
- Es wäre also fatal, wenn ein solches Programm sehr lange laufen würde oder sogar in einer Endlosschleife stecken bliebe.
- Ein Ausweg besteht darin, die Programme so stark einzuschränken, dass Endlosschleifen nicht möglich sind. Der Nachteil dabei ist, dass dies die Menge der möglichen Programme einschränkt.
- Eine andere Möglichkeit besteht darin, beliebig komplexe Programme zuzulassen, aber ihre Ausführung “teuer” zu machen, d.h. der Autor einer Transaktion muss für jeden Schritt, den das Programm macht, mittels Transaktionsgebühren bezahlen.
- Bitcoin hat sich für den ersten Weg entschieden, Ethereum und Cardano für den zweiten.

- Programmiersprachen können nach ihrer Turing-Vollständigkeit (Turing Completeness) klassifiziert werden.
- Eine Programmiersprache heisst Turing complete (Turing-vollständig), wenn man in ihr jede beliebige Turing-Maschine simulieren kann.
- Intuitiv bedeutet dies, dass man in solchen Programmiersprachen alles berechnen kann, was berechenbar ist.
- Da Turing-Maschinen in Endlosschleifen geraten können, gilt dies insbesondere auch für Turing-vollständige Programmiersprachen.
- In der Praxis sind alle gängigen höheren Programmiersprachen Turing-vollständig: Python, Java, C, C++, Perl, Javascript, Lisp, Haskell,...
- Auch viele exotischere Konstruktionen (z.B. Lambda-Kalkül) sind Turing-vollständig.

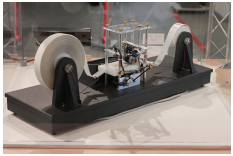


Abb.: Turing-Maschine, rekonstruiert von Mike Davey. Foto von Rocky Acosta, Creative Commons Lizenz.

- Das berühmte Halte-Problem ist die Frage, ob man einem Programm, das in einer Turing-vollständigen Sprache geschrieben ist, “ansehen” kann, ob es in eine Endlosschleife geraten wird.
- Genauer: Gibt es ein Programm, dass für jedes Programm entscheidet, ob dieses halten wird?
- Die Antwort ist nein! — Angenommen, es gäbe eine Python-Funktion `halt`, die das Halteproblem löst. Man betrachte dann die folgende Python-Funktion:

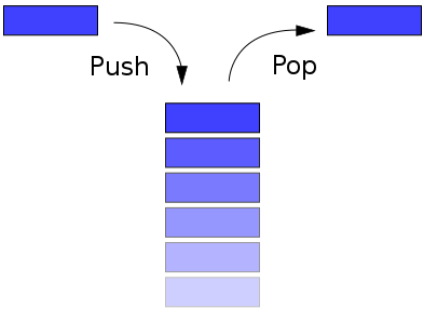
```
def paradox():
    if halt(paradox):
        while True:
            pass
```

Falls `halt True` zurück gibt, gerät `paradox` in eine Endlosschleife, was ein Widerspruch ist. Falls `halt False` zurück gibt, endet `paradox`, was ebenfalls ein Widerspruch ist.

- Wegen des Halte-Problems ist es unmöglich, automatisch zu entscheiden, ob ein gegebenes Programm in einer Turing-vollständigen Sprache irgendwann anhalten wird oder nicht.
- Wenn man also eine Turing-vollständige Smart-Contract-Sprache wählt, kann man vorab nicht garantieren, dass ein Skript anhalten wird. Man kann auch nicht absehen, wie lange es laufen wird.
- Daher muss man sich entweder gegen Turing-Vollständigkeit entscheiden oder Programme nach endlicher Zeit abbrechen.

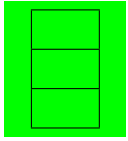
- Bitcoin benutzt eine Sprache, die nicht Turing-vollständig ist und Bitcoin Script genannt wird.
- Bitcoin Script enthält keine Schleifen, d.h. ein Programm in Bitcoin Script kann nie in eine Endlosschleife geraten.
- Trotz seiner Einfachheit ist Bitcoin Script recht mächtig und flexibel und ermöglicht vielfältige Arten von Transaktions-Verifizierungen.
- Andererseits ist Bitcoin Script zu eingeschränkt, um echte Smart Contracts im Sinne von komplexen finanziellen Verträgen abzubilden.

- Bitcoin Script ist eine sogenannte Stack based (auf Stapeln basierende) Sprache.
- Forth ist eine (relativ) populäre höhere Programmiersprache, die ebenfalls auf Stapeln basiert. Andere Beispiele sind die Java Virtual Machine (JVM) und die Common Language Runtime (CLR) von Microsoft.
- In Bitcoin Script gibt es keine Variablen. Daten werden stattdessen auf dem Stapel abgelegt und verarbeitet.



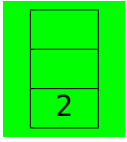
- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
2
3
op_add
4
op_mul
```



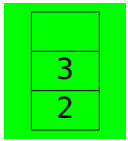
- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

- 2
  3
  op_add
  4
  op_mul



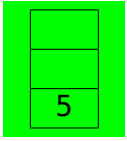
- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
  2
• 3
  op_add
  4
  op_mul
```



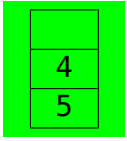
- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
  2
  3
• op_add
  4
  op_mul
```



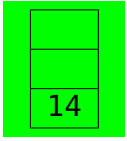
- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
  2
  3
  op_add
• 4
  op_mul
```



- Das folgende Programm in Bitcoin Script berechnet $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
  2
  3
  op_add
  4
• op_mul
```



- Berechnen Sie $(10 - 3) \cdot (4 + 7)$ mit Bitcoin Script! (*Hinweis*: Benutzen Sie `op_sub`!)
- Schreiben Sie ein Bitcoin Script Programm, das die Zahl auf dem Stack quadriert. (*Hinweis*: Benutzen Sie `op_dup`!)
- Schreiben Sie ein Bitcoin Script Programm, das $x^2 + y^2$ berechnet, wenn x und y die beiden obersten Zahlen auf dem Stapel sind. (*Hinweis*: Benutzen Sie `op_swap`!)
- Schreiben Sie ein Bitcoin Script Programm, das $x \cdot y$ berechnet, falls $y < x$, und $x + y$, falls $y >= x$, wobei x und y die beiden oberen Zahlen auf dem Stack sind (x oben, y an zweiter Stelle). (*Hinweis*: Benutzen Sie `op_2dup`, `op_lessthan`, `op_if`, `op_else`, und `op_endif`!)
- Sie finden eine Liste aller Bitcoin-Script Befehle auf `https://en.bitcoin.it/wiki/Script`.
- Auf `https://siminchen.github.io/bitcoinIDE/build/editor.html` gibt es einen hübschen Online-Simulator.

- Bei Bitcoin enthalten sowohl Transaktions-Outputs als auch Transaktions-Inputs Skripte.
- Zum Verifizieren einer Transaktion wird jeder Input wie folgt geprüft:
  - Der Output-Skript wird mit dem Input-Skript kombiniert (erst Output-Skript, dann Input-Skript).
  - Diese Kombination wird ausgeführt.
  - Die Benutzung des Inputs ist gültig, wenn der Skript keinen Fehler meldet und wenn am Ende eine Zahl auf dem Stapel liegt, die nicht Null ist.
- Die Transaktion ist gültig, wenn alle Inputs auf diese Weise gültig sind (und wenn alle anderen Bedingungen erfüllt sind, also Summe der Inputs größer Summe der Outputs usw.).

- Die allermeisten Bitcoin-Transaktionen benutzen gewöhnliche “Hash eines öffentlichen Schlüssels”-Adressen.
- Wie sehen Input- und Output-Skripte in diesem Fall aus?
  - `<sig> <pubKey>`
  - `op_dup op_hash160 <pubKeyHash> op_equalverify op_checksig`
  - Der Input-Skript legt die elektronische Unterschrift und den öffentlichen Schlüssel auf den Stapel. Der Output-Skript prüft, ob der Hash dieses öffentlichen Schlüssels den richtigen Wert hat und ob die Unterschrift korrekt ist.

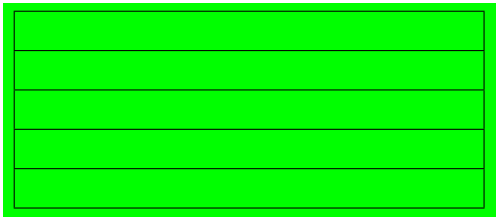
- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

```
304402200cc8b0471a38ed...
02ce9f5972fe1473c9b694...
op_dup
op_hash160
1290b657a78e201967c22d...
op_equalverify
op_checksig
```



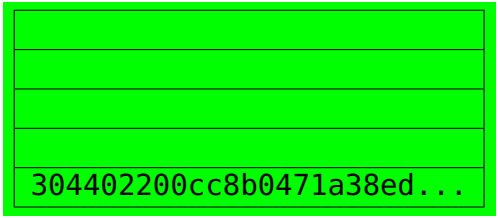
- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

- 304402200cc8b0471a38ed...
  02ce9f5972fe1473c9b694...
  op_dup
  op_hash160
  1290b657a78e201967c22d...
  op_equalverify
  op_checksig

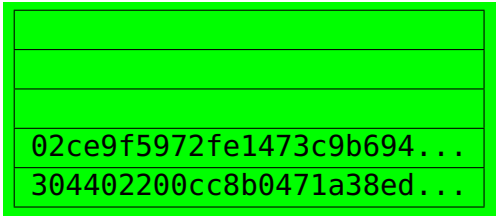


- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

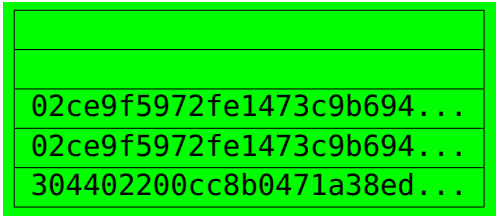


- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

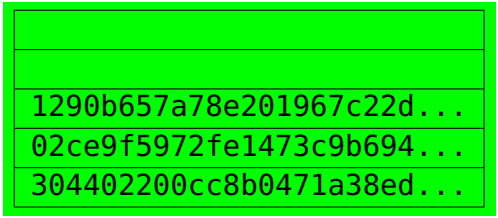


- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

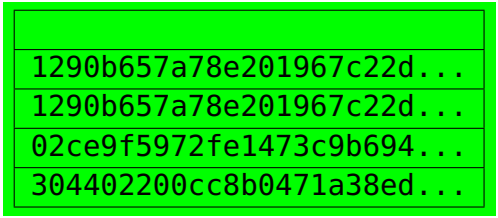


- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

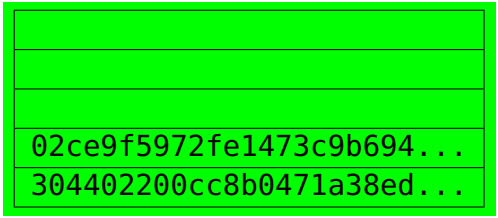


- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```



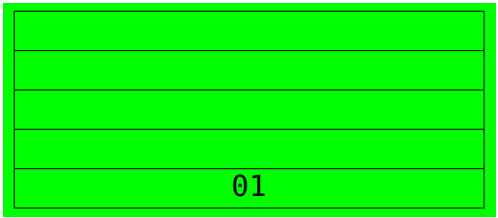
- Input-Skript:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Skript:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...
02ce9f5972fe1473c9b694...
op_dup
op_hash160
1290b657a78e201967c22d...
op_equalverify
• op_checksig



- Ein anderer häufiger Transaktionstyp in Bitcoin ist Multisig.
- Bei einer solchen Transaktion müssen mehrere Parteien digital unterschreiben, um einen Output ausgeben zu dürfen.
- Bitcoin Script unterstützt dies mittels `op_checkmultisig`.

Andere Bitcoin Script Typen — Unausgebbare Transaktionen

- Der Script-Befehl `op_return` macht einen Output unausgebbar.
- Alle Befehle nach `op_return` werden ignoriert.
- Eine Anwendung ist z.B., einen Output vom Wert Null zu erzeugen und ihm einen Skript zu geben, der mit `op_return` beginnt und dann beliebige Daten enthält.

- Der Output von Transaktion `a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b` war mit folgendem Skript versehen:

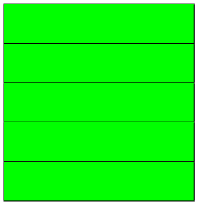
```
op_hash256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
op_equal
```

- Um den Output auszugeben, musste man also eine Zahl finden, die den gegebenen Hash hat.
- Es gelang tatsächlich jemandem, diese Rätsel zu lösen: Die gesuchte Zahl war der Hash des Genesis-Block-Headers.

- Im Jahr 2013 kreierte Peter Todd Skripte, deren Outputs von jedem ausgegeben werden können, der eine SHA-1-Hash-Kollision findet:
  ```
  op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal
  ```
- Man kann Bitcoin an so gesicherte Skript-Adressen schicken, um einen Anreiz zu bieten, Hash-Kollisionen zu finden.
- Solange die Belohnung hinreichend groß war und niemand sie gewonnen hatte, konnte man relativ sicher sein, dass niemand eine Kollision gefunden hatte.
- Im Februar 2017 wurde die Belohnung von 2,48 ₿ von jemandem gewonnen.

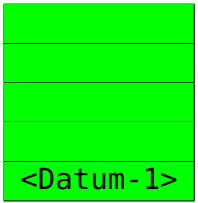
Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal
```



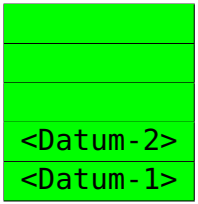
- <Datum-1>

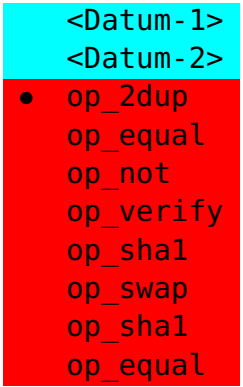
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal

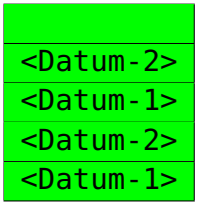


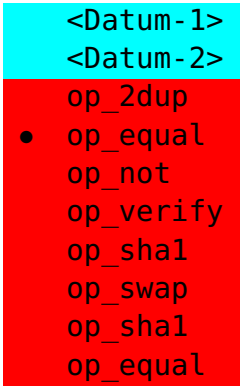
Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen

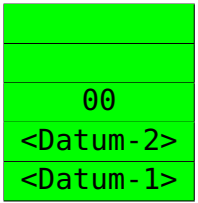
```
  <Datum-1>
• <Datum-2>
  op_2dup
  op_equal
  op_not
  op_verify
  op_sha1
  op_swap
  op_sha1
  op_equal
```



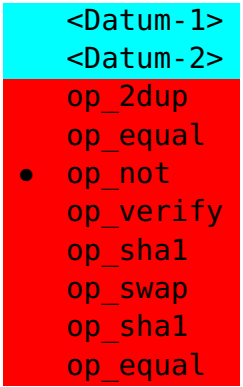

<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal

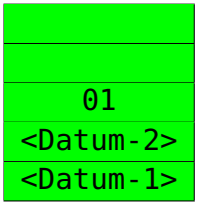

<Datum-2>
<Datum-1>
<Datum-2>
<Datum-1>


<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal

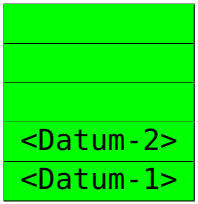

00
<Datum-2>
<Datum-1>

Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen



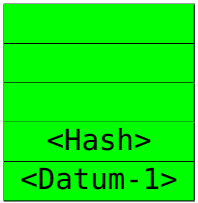


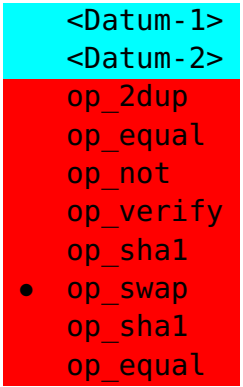
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
• op_verify
op_sha1
op_swap
op_sha1
op_equal

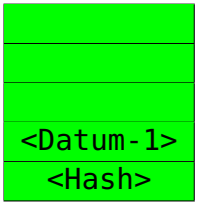


Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen

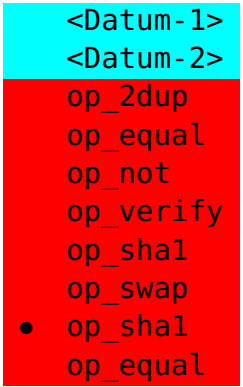
```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal
```

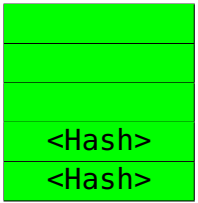



<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal


<Datum-1>
<Hash>

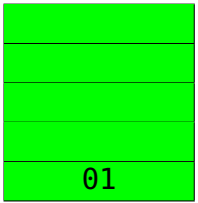
Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen





Andere Bitcoin Script Typen — Anreize für Hash-Kollisionen

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal
```



- Bitcoin Script unterstützt Zeitschlösser (Time Locks) mittels `op_checklocktimeverify` und `op_checksequenceverify`.
- `op_checklocktimeverify` ermöglicht das Ausgeben eines Outputs erst, wenn ein bestimmter Zeitpunkt (gemessen in Blockhöhe) erreicht ist.
- `op_checksequenceverify` basiert im Gegensatz dazu auf relativer Zeit: Die Ausgabe ist erst möglich, nachdem die Transaktion eine gewisse Tiefe in der Blockchain erreicht hat.
- Beide Typen von Zeitschlössern finden z.B. in Bitcoin Lightning Anwendung.

Zeitschloss Beispiel

- Alice und Bob sind Geschäftspartner und sichern ihr Geld mittels 2-von-2-Multisig (d.h. beide müssen unterschreiben).
- Sie haben Angst, dass ihr Geld verloren ist, wenn einem der beiden etwas passiert.
- Sie beschließen daher, ihren gemeinsamen Anwalt Charlie um Hilfe zu bitten.
- Würden sie eine gewöhnliche 2-von-3-Multisig benutzen, könnten Alice oder Bob gemeinsam mit Charlie das Geld stehlen.
- Stattdessen können Sie das folgende Skript benutzen:

```
op_if <in drei Monaten> op_checklocktimeverify op_drop <PubKey-Charlie>
op_checksigverify 1 op_else 2 op_endif <PubKey-Alice> <PubKey-Bob> 2
op_checkmultisig
```

- Alice und Bob gemeinsam können jederzeit an ihr Geld, wenn sie folgendes Skript benutzen:

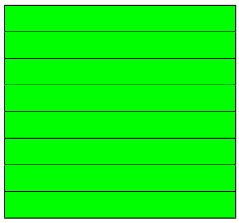
```
0 <Sig-Alice> <Sig-Bob> 0
```

- Nach drei Monaten können Charlie und entweder Alice oder Bob folgendes Skript benutzen:

```
0 <Sig-Alice/Bob> <Sig-Charlie> 1
```

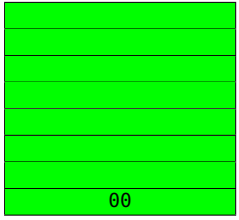
Zeitschloss Beispiel — erster Fall

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



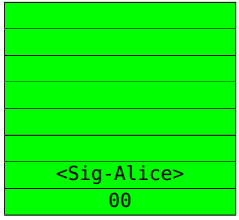
Zeitschloss Beispiel — erster Fall

- 00
  <Sig-Alice>
  <Sig-Bob>
  00
  op_if
  <in drei Monaten>
  op_checklocktimeverify
  op_drop
  <PubKey-Charlie>
  op_checksigverify
  01
  op_else
  02
  op_endif
  <PubKey-Alice>
  <PubKey-Bob>
  02
  op_checkmultisig

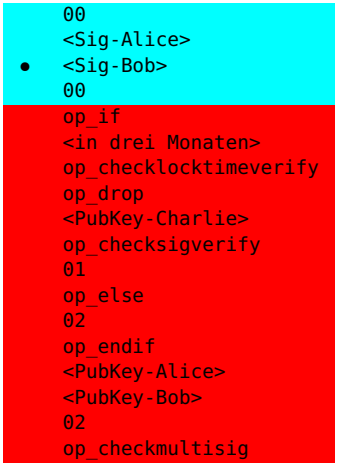


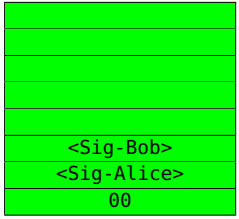
Zeitschloss Beispiel — erster Fall

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

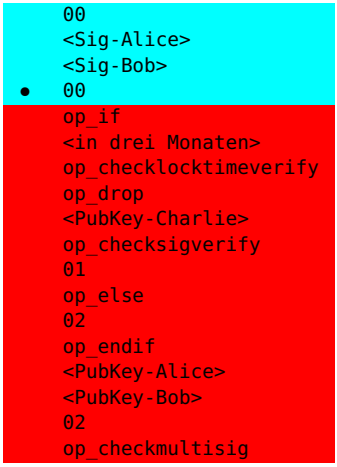


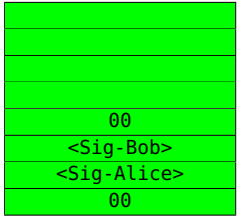
Zeitschloss Beispiel — erster Fall





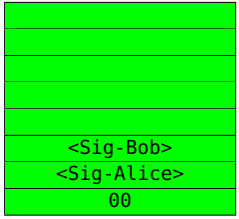
Zeitschloss Beispiel — erster Fall



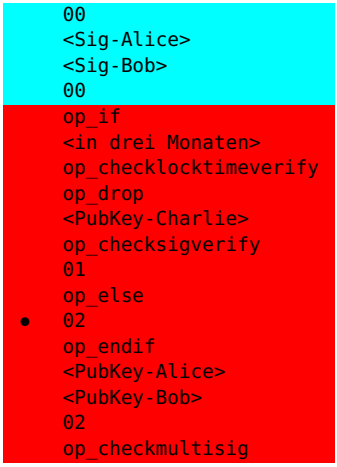


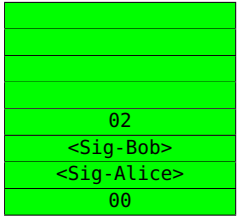
Zeitschloss Beispiel — erster Fall

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

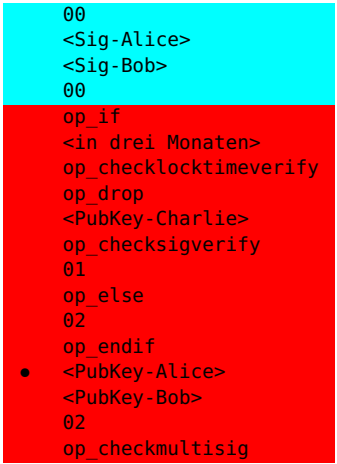


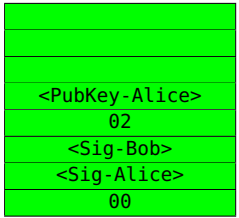
Zeitschloss Beispiel — erster Fall



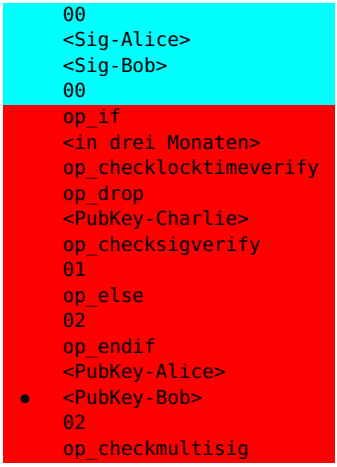


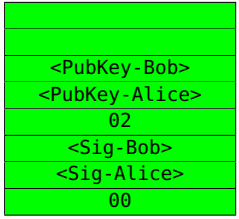
Zeitschloss Beispiel — erster Fall



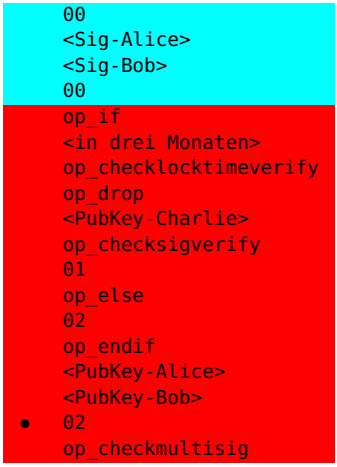


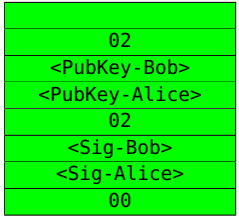
Zeitschloss Beispiel — erster Fall





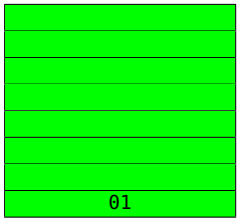
Zeitschloss Beispiel — erster Fall





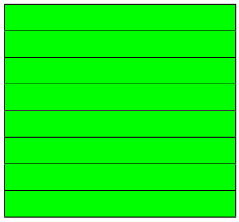
Zeitschloss Beispiel — erster Fall

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



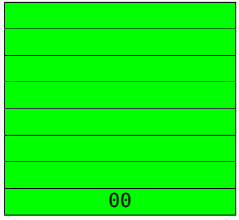
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



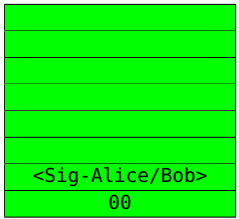
Zeitschloss Beispiel — zweiter Fall

- 00
  <Sig-Alice/Bob>
  <Sig-Charlie>
  01
  op_if
  <in drei Monaten>
  op_checklocktimeverify
  op_drop
  <PubKey-Charlie>
  op_checksigverify
  01
  op_else
  02
  op_endif
  <PubKey-Alice>
  <PubKey-Bob>
  02
  op_checkmultisig

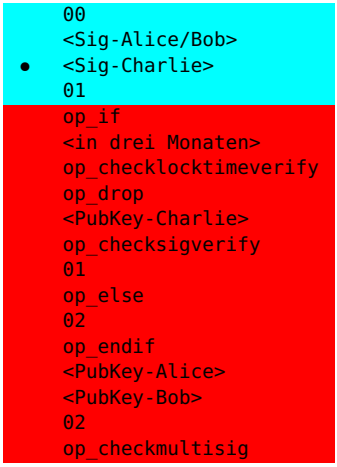


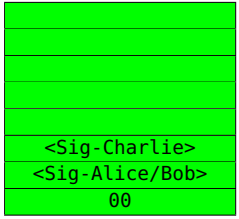
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

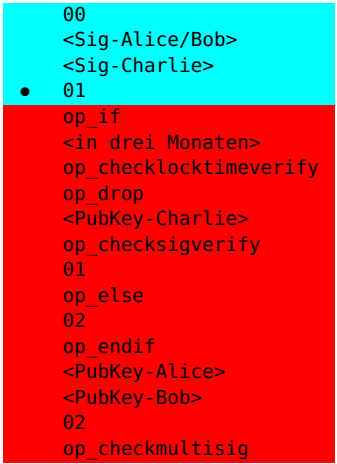


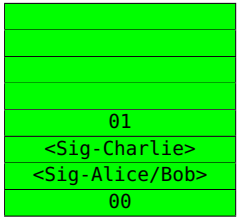
Zeitschloss Beispiel — zweiter Fall





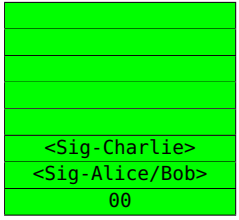
Zeitschloss Beispiel — zweiter Fall





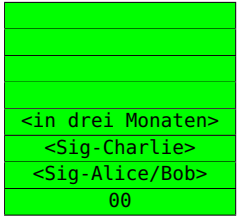
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



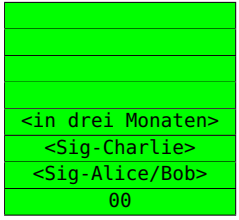
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



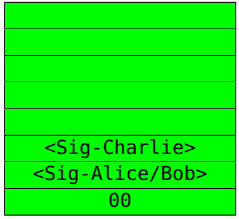
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



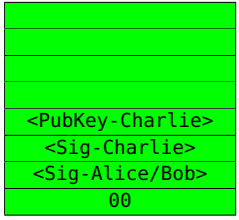
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



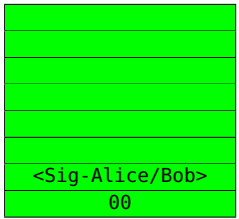
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



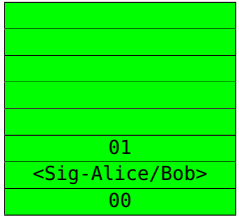
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



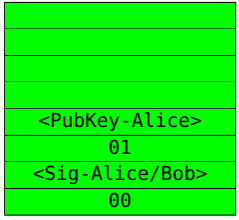
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

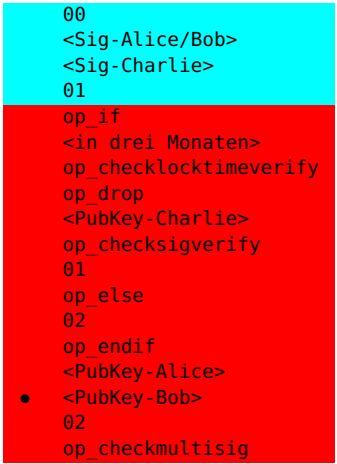


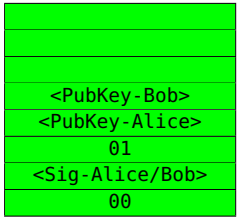
Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

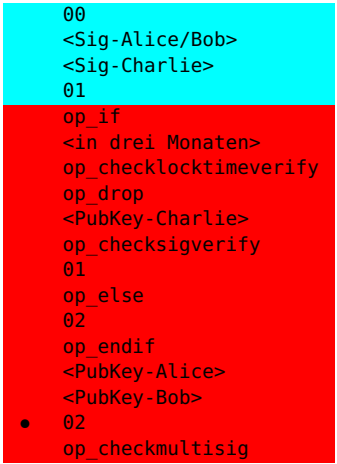


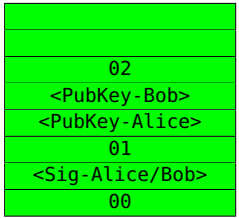
Zeitschloss Beispiel — zweiter Fall





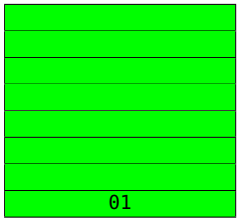
Zeitschloss Beispiel — zweiter Fall





Zeitschloss Beispiel — zweiter Fall

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



Hinweis

Diese Publikation wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Bund- Länder- Wettbewerbs “Aufstieg durch Bildung: offene Hochschulen” erstellt. Die in dieser Publikation dargelegten Ergebnisse und Interpretationen liegen in der alleinigen Verantwortung der Autor/innen.