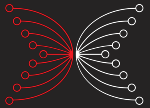
Smart Contracts

Smart Contracts & Bitcoin-Script

Lars Brünjes



INPUT | OUTPUT

January 9 2020

Reminder: The Simple UTxO-Model

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.
- As in the account-based model (employed for example by Ethereum), the UTxO-model uses (hashes of) public keys as addresses.

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.
- As in the account-based model (employed for example by Ethereum), the UTxO-model uses (hashes of) public keys as addresses.
- A transaction has UTxOs as inputs and one or more outputs.

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.
- As in the account-based model (employed for example by Ethereum), the UTxO-model uses (hashes of) public keys as addresses.
- A transaction has UTxOs as inputs and one or more outputs.
- Transactions are authorized by the digital signatures of the owners of the inputs.

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.
- As in the account-based model (employed for example by Ethereum), the UTxO-model uses (hashes of) public keys as addresses.
- A transaction has UTxOs as inputs and one or more outputs.
- Transactions are authorized by the digital signatures of the owners of the inputs.
- Transactions with several senders and/or receivers are possible.

The Simple UTxO-Model

- UTxO is an acronym for Unspent Transaction Output.
- As in the account-based model (employed for example by Ethereum), the UTxO-model uses (hashes of) public keys as addresses.
- A transaction has UTxOs as inputs and one or more outputs.
- Transactions are authorized by the digital signatures of the owners of the inputs.
- Transactions with several senders and/or receivers are possible.
- The state of the blockchain is determined by the set of all UTxOs.

The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus transaction fees) must equal the sum of all outputs.

The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus transaction fees) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.

The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus transaction fees) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
- A transaction consists of:
  - A set of inputs (UTxOs).

The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus transaction fees) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
- A transaction consists of:
  - A set of inputs (UTxOs).
  - An ordered list of outputs, where each output has an address and a value.

The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus transaction fees) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
- A transaction consists of:
  - A set of inputs (UTxOs).
  - An ordered list of outputs, where each output has an address and a value.
  - A digital signature for each input.

Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is valid if the following three conditions are satisfied:

Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is valid if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.

Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is valid if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.
- The sum of all input values (plus transaction fees) equals the sum of all outputs.

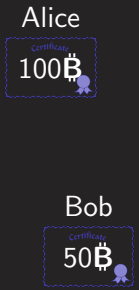
Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is valid if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.
- The sum of all input values (plus transaction fees) equals the sum of all outputs.
- No output value is negative.

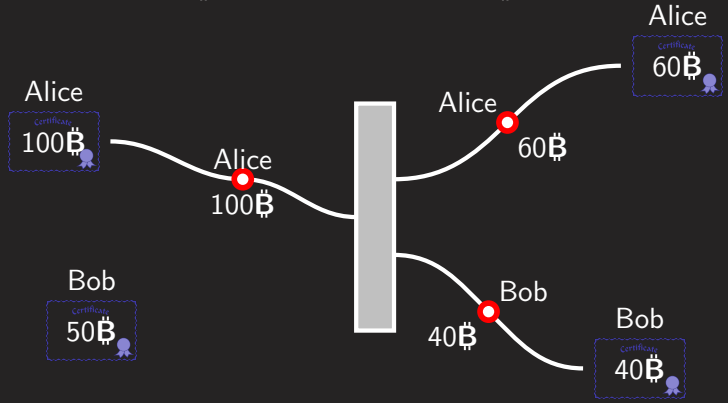
Example: A Simple Transaction

Alice holds 100 ฿ and wants to send 40 ฿ to Bob, who has 50 ฿.



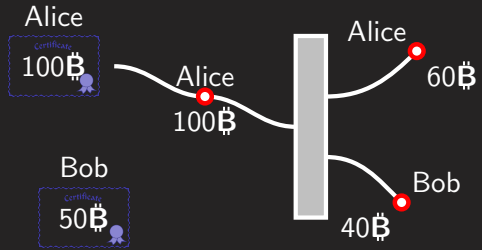
Example: A Simple Transaction

Alice holds 100 ₿ and wants to send 40 ₿ to Bob, who has 50 ₿.



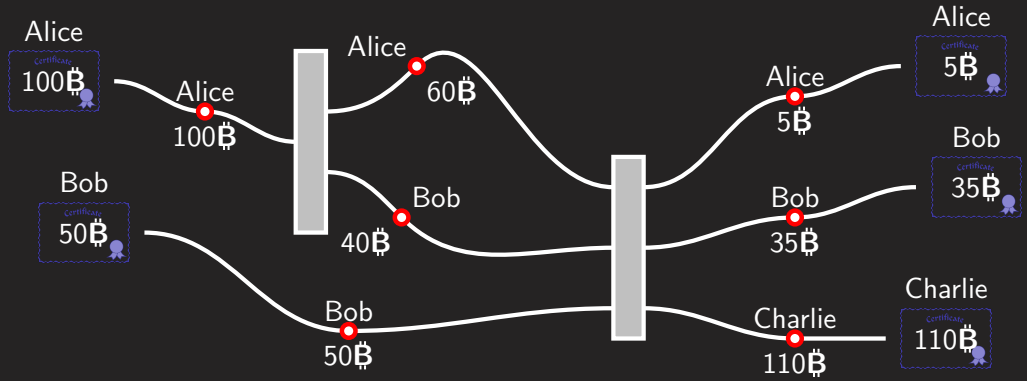
Example: A More Complex Transaction

After sending 40 ₿ to Bob, Alice and Bob want to send 55 ₿ each to Charlie.



Example: A More Complex Transaction

After sending 40 ₿ to Bob, Alice and Bob want to send 55 ₿ each to Charlie.



Reminder: Smart Contracts

Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity “on the blockchain”.

Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity "on the blockchain".
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).

Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity "on the blockchain".
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
- Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.

Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity "on the blockchain".
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
- Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.
- Whether this is good or bad is up to discussion: Courts and human judges are fallible, but they are also able to use common sense to see the *intent* of a contract without sticking to the words.

Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity "on the blockchain".
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
- Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.
- Whether this is good or bad is up to discussion: Courts and human judges are fallible, but they are also able to use common sense to see the *intent* of a contract without sticking to the words.
- In this sense, smart contracts are *only* "words": Intent does not matter. All that matters is the actual code.

Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, addresses (which identify senders and receivers) are simply (hashes of) public keys.

Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, addresses (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.

Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, addresses (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called scripts.

Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, addresses (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called scripts.
- While verifying the inputs of a Bitcoin transaction, input- and output-scripts are combined and executed. The result from this execution decides whether the spending transaction is entitled to spend the output.

Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, addresses (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called scripts.
- While verifying the inputs of a Bitcoin transaction, input- and output-scripts are combined and executed. The result from this execution decides whether the spending transaction is entitled to spend the output.
- Details depend on the specific cryptocurrency, but the principle stays the same: Programs decide under which circumstances money may be spent.

Flavours of Smart Contracts

Dilemma

- The programs which decide transaction validity must run on all nodes.

Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.

Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
- One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.

Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
- One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.
- Another option is to allow arbitrarily complex programs, but to make their execution “expensive”: The initiator of a transaction has to pay a fee for each step the programs takes.

Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
- One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.
- Another option is to allow arbitrarily complex programs, but to make their execution "expensive": The initiator of a transaction has to pay a fee for each step the programs takes.
- The creators of Bitcoin have chosen the first option, Ethereum and Cardano use the second.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.
- A programming language is Turing-complete if every Turing-machine can be simulated in it.

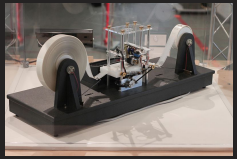


Figure: Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.
- A programming language is Turing-complete if every Turing-machine can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.

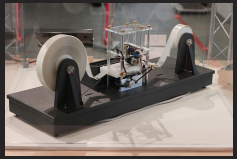


Figure: Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.
- A programming language is Turing-complete if every Turing-machine can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.

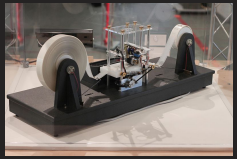


Figure: Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.
- A programming language is Turing-complete if every Turing-machine can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.
- All commonly used "higher" programming languages are Turing-complete: Python, Java, C, C++, Perl, JavaScript, Lisp, Haskell,...

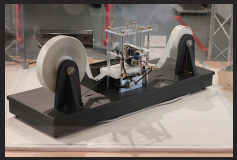


Figure: Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

Interlude: Turing-Completeness

- Programming languages can be Turing-complete or not.
- A programming language is Turing-complete if every Turing-machine can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.
- All commonly used "higher" programming languages are Turing-complete: Python, Java, C, C++, Perl, JavaScript, Lisp, Haskell,...
- Many more exotic systems (like λ-calculus) are Turing-complete as well.

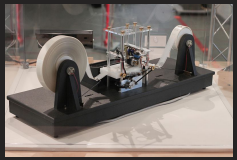


Figure: Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

Interlude: The Halting-Problem

- The famous Halting-Problem considers the question whether one can "see" if a program written in a Turing-complete language will get stuck in an infinite loop or halt.

Interlude: The Halting-Problem

- The famous Halting-Problem considers the question whether one can "see" if a program written in a Turing-complete language will get stuck in an infinite loop or halt.
- To be more precise: Is there a program that will take an arbitrary program as input and then decide whether that program will halt?

Interlude: The Halting-Problem

- The famous Halting-Problem considers the question whether one can "see" if a program written in a Turing-complete language will get stuck in an infinite loop or halt.
- To be more precise: Is there a program that will take an arbitrary program as input and then decide whether that program will halt?
- The answer is no! — Assume there was a Python-function `halt` solving the Halting-Problem. Then consider the following Python-function:

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

If `halt` returns `True`, `paradox` gets stuck in an infinite loop, and if `halt` returns `False`, `paradox` will halt. Both are contradictions.

Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.

Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.
- So if one chooses a Turing-complete smart-contract language, it is impossible to guarantee in advance whether a script will stop. Nor is it possible to know for how long the script will run, even if it eventually halts.

Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.
- So if one chooses a Turing-complete smart-contract language, it is impossible to guarantee in advance whether a script will stop. Nor is it possible to know for how long the script will run, even if it eventually halts.
- As a consequence, one either has to decide against using a Turing-complete language or be prepared to interrupt a running script after some finite time has passed.

Bitcoin Script

Bitcoin Script

- Bitcoin uses a language which is not Turing-complete and which is called Bitcoin Script.

Bitcoin Script

- Bitcoin uses a language which is not Turing-complete and which is called Bitcoin Script.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.

Bitcoin Script

- Bitcoin uses a language which is not Turing-complete and which is called Bitcoin Script.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.
- In spite of its simplicity, Bitcoin Script is quite powerful and flexible and allows for a plethora of different kinds of transaction verification.

Bitcoin Script

- Bitcoin uses a language which is not Turing-complete and which is called Bitcoin Script.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.
- In spite of its simplicity, Bitcoin Script is quite powerful and flexible and allows for a plethora of different kinds of transaction verification.
- On the other hand, Bitcoin Script is too limited to allow for real smart contracts implementing complex financial transactions.

Stacks

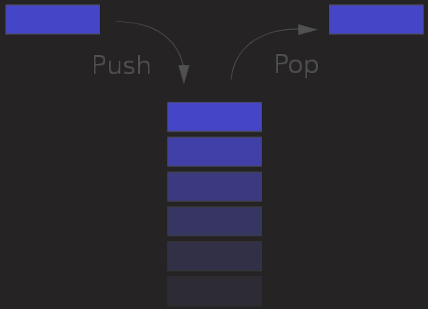
- Bitcoin Script is a so-called Stack based language.

Stacks

- Bitcoin Script is a so-called Stack based language.
- Forth is a (relatively) popular higher programming language that is stack based as well. Other examples are the Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR).

Stacks

- Bitcoin Script is a so-called Stack based language.
- Forth is a (relatively) popular higher programming language that is stack based as well. Other examples are the Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR).
- There are no variables in Bitcoin Script. Instead, data is put onto the stack and processed there.



Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.

Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).

Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.

Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.
  - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.

Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.
  - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.
- The transaction is valid if all inputs are valid in this sense (and if all other conditions are satisfied, so the sum of all input-values is greater than the sum of all output-values etc.).

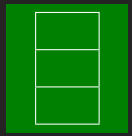
Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.
  - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.
- The transaction is valid if all inputs are valid in this sense (and if all other conditions are satisfied, so the sum of all input-values is greater than the sum of all output-values etc.).
- This script-mechanism extends the simple UTxO-model by allowing for more complex input-validation, going beyond digital signature verification.

Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

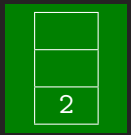
```
2
3
op_add
4
op_mul
```



Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

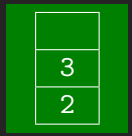
- ```
  2
  3
  op_add
  4
  op_mul
  ```


  ```

Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

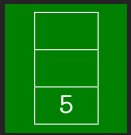
```
2
3
op_add
4
op_mul
```



Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

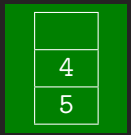
```
  2
  3
• op_add
  4
  op_mul
```



Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

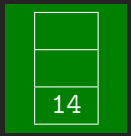
```
2
3
op_add
4
op_mul
```



Example: Arithmetic

- The following program written in Bitcoin Script calculates $(2 + 3) \cdot 4$:
- `2 3 op_add 4 op_mul`

```
  2
  3
  op_add
  4
• op_mul
```



Exercises

- Compute $(10 - 3) \cdot (4 + 7)$ using Bitcoin Script! (*Hint*: Use `op_sub`!)
- Write a Bitcoin Script program which squares the number on top of the stack. (*Hint*: Use `op_dup`!)
- Write a Bitcoin Script programm which computes $x^2 + y^2$, where x and y are the two top-most numbers on the stack. (*Hint*: Use `op_swap`!)
- Write a Bitcoin Script Programm which computes $x \cdot y$ if $y < x$ and $x + y$ if $y >= x$, where x and y are the two top-most numbers on the stack (x on top, y below). (*Hint*: Use `op_2dup`, `op_lessthan`, `op_if`, `op_else`, and `op_endif`!)
- You can find a list of all Bitcoin-Script commands on `https://en.bitcoin.it/wiki/Script`.
- There is a nice online simulator on `https://siminchen.github.io/bitcoinIDE/build/editor.html`.

Pay to Public Key Hash in Bitcoin Script

- The vast majority of all Bitcoin transaction uses ordinary “hash of public key”-addresses.
- What do input- and output-scripts look like in this case?
  - `<sig> <pubKey>`
  - `op_dup op_hash160 <pubKeyHash> op_equalverify op_checksig`
  - The input-script puts the digital signature and the public key onto the stack. The output-script checks whether the hash of this public key has the right value and whether the signature is correct.

Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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

Pay to Public Key Hash — Example

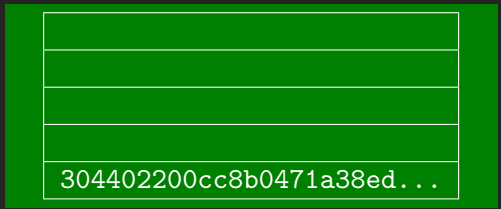
- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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



Pay to Public Key Hash — Example

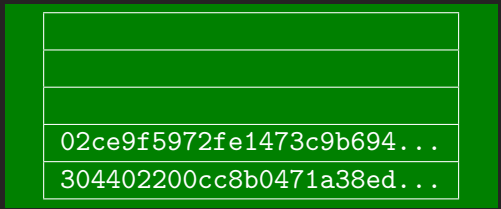
- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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



Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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

02ce9f5972fe1473c9b694...
02ce9f5972fe1473c9b694...
304402200cc8b0471a38ed...

Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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

1290b657a78e201967c22d...
02ce9f5972fe1473c9b694...
304402200cc8b0471a38ed...

Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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

1290b657a78e201967c22d...
1290b657a78e201967c22d...
02ce9f5972fe1473c9b694...
304402200cc8b0471a38ed...

Pay to Public Key Hash — Example

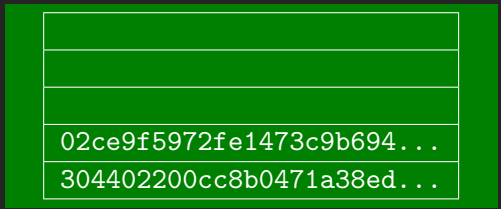
- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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



Pay to Public Key Hash — Example

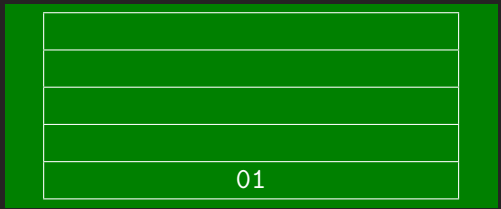
- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceefd8dc
```

- Output-Script:

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

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



Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is multisig.

Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is multisig.
- If an output uses multisig, it can only be spent if several parties provide their signatures.

Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is multisig.
- If an output uses multisig, it can only be spent if several parties provide their signatures.
- Bitcoin Script supports this via `op_checkmultisig`.

Other kinds of Bitcoin Scripts — Unspendable Outputs

- The scipt-command `op_return` renders an output unspendable.

Other kinds of Bitcoin Scripts — Unspendable Outputs

- The scipt-command `op_return` renders an output unspendable.
- All commands following `op_return` are ignored.

Other kinds of Bitcoin Scripts — Unspendable Outputs

- The scipt-command `op_return` renders an output unspendable.
- All commands following `op_return` are ignored.
- One possible application of this is to for example generate an output with value zero, then use a script which starts with `op_return` and contains arbitrary data afterwards.

Other kinds of Bitcoin Scripts — Riddles

- The output of transaction `a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b` contained the following script:

```
op_hash256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
op_equal
```

Other kinds of Bitcoin Scripts — Riddles

- The output of transaction `a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b` contained the following script:

```
op_hash256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
op_equal
```

- To spend that output, one had to find a number with the given hash.

Other kinds of Bitcoin Scripts — Riddles

- The output of transaction `a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b` contained the following script:

```
op_hash256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
op_equal
```

- To spend that output, one had to find a number with the given hash.
- This riddle was eventually solved: The given hash turned out to be the hash of the genesis-block-header.

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spend by anybody who found a hash collision for SHA-1:

```
op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal
```

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spend by anybody who found a hash collision for SHA-1:
  `op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

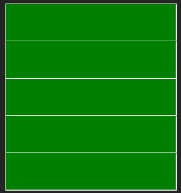
- In the year 2013, Peter Todd created scripts whose outputs could be spend by anybody who found a hash collision for SHA-1:
  `op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.
- As long as the reward was high enough and nobody claimed it, one could be relatively certain that SHA-1 was still safe and that nobody had found a collision.

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spend by anybody who found a hash collision for SHA-1:
  `op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.
- As long as the reward was high enough and nobody claimed it, one could be relatively certain that SHA-1 was still safe and that nobody had found a collision.
- In February 2017, somebody claimed the reward of 2.48 ฿.

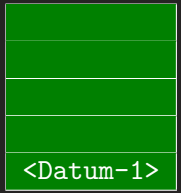
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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



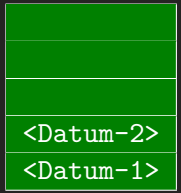
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

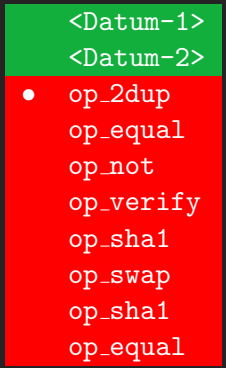


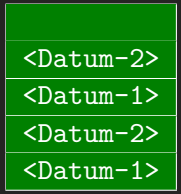
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

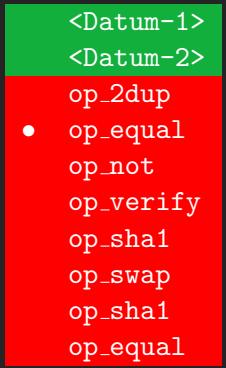


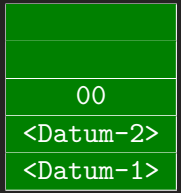
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions



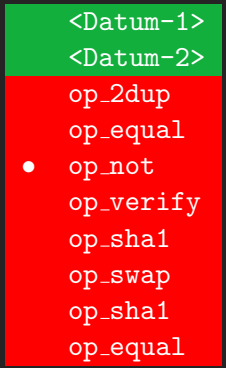


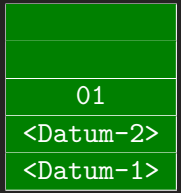
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions





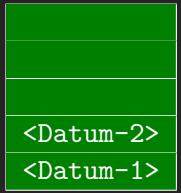
Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions





Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

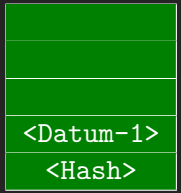


Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

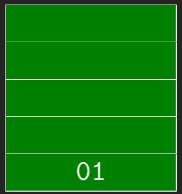


Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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

Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

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



Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports time locks via `op_checklocktimeverify` and `op_checksequenceverify`.

Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports time locks via `op_checklocktimeverify` and `op_checksequenceverify`.
- `op_checklocktimeverify` only allows spending an output when a certain point in time (measured in block height) has been reached.

Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports time locks via `op_checklocktimeverify` and `op_checksequenceverify`.
- `op_checklocktimeverify` only allows spending an output when a certain point in time (measured in block height) has been reached.
- `op_checksequenceverify`, on the other hand, uses relative time: Spending the output is only possible after the transaction has reached a given depth in the blockchain.

Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports time locks via `op_checklocktimeverify` and `op_checksequenceverify`.
- `op_checklocktimeverify` only allows spending an output when a certain point in time (measured in block height) has been reached.
- `op_checksequenceverify`, on the other hand, uses relative time: Spending the output is only possible after the transaction has reached a given depth in the blockchain.
- Both types of time locks are for example used in Bitcoin Lightning.

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
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
```

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
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 and Bob can access their money at any time using this script:

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

Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
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 and Bob can access their money at any time using this script:

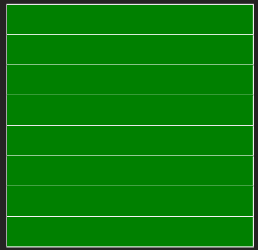
```
0 <Sig-Alice> <Sig-Bob> 0
```

- After three months, Charlie and either Alice or Bob can use the following script instead:

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

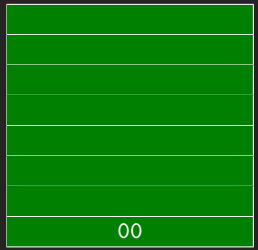
Time Lock Example — First Case

```
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
```



Time Lock Example — First Case

- 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

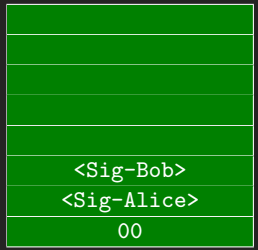


Time Lock Example — First Case

```
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
```

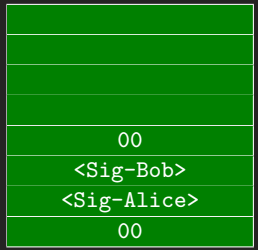
Time Lock Example — First Case

```
   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
```



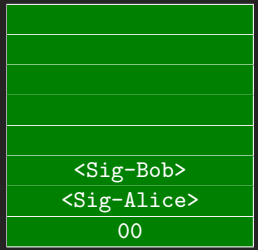
Time Lock Example — First Case

```
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
```



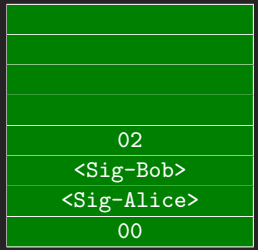
Time Lock Example — First Case

```
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
```



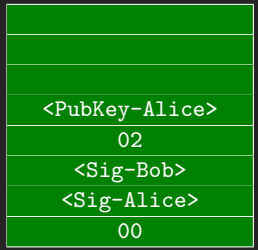
Time Lock Example — First Case

```
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
```



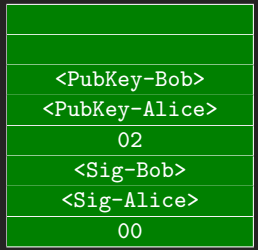
Time Lock Example — First Case

```
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
```



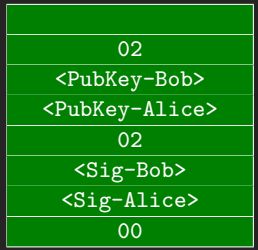
Time Lock Example — First Case

```
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
```



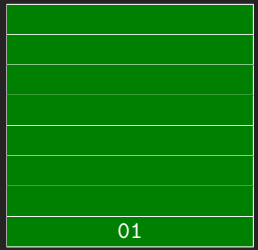
Time Lock Example — First Case

```
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
```



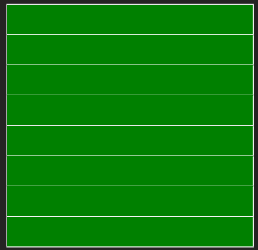
Time Lock Example — First Case

```
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
```



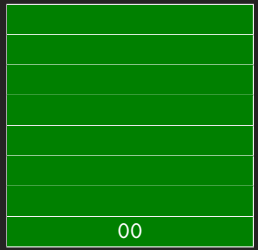
Time Lock Example — Second Case

```
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
```



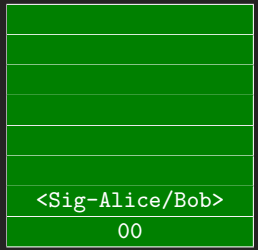
Time Lock Example — Second Case

- 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



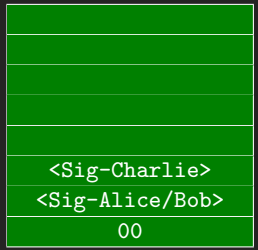
Time Lock Example — Second Case

```
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
```



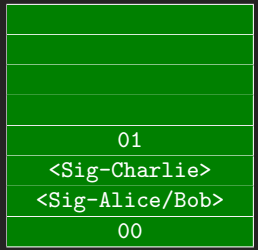
Time Lock Example — Second Case

```
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
```



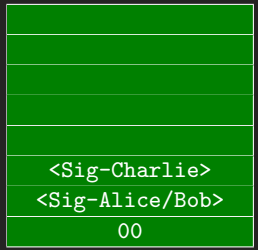
Time Lock Example — Second Case

```
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
```



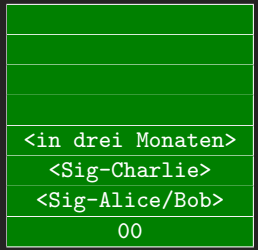
Time Lock Example — Second Case

```
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
```



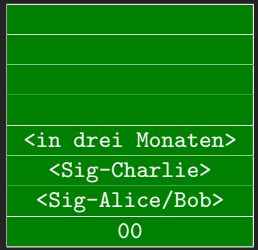
Time Lock Example — Second Case

```
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
```



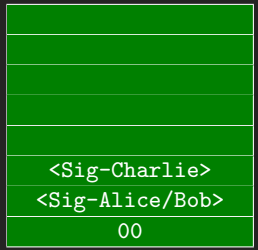
Time Lock Example — Second Case

```
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
```



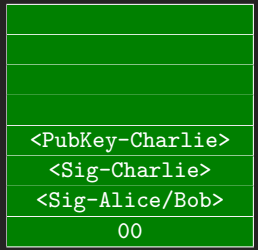
Time Lock Example — Second Case

```
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
```



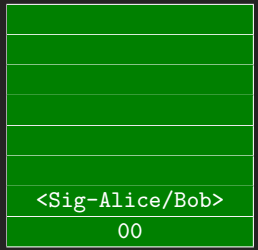
Time Lock Example — Second Case

```
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
```



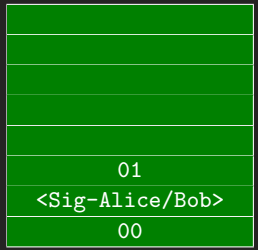
Time Lock Example — Second Case

```
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
```



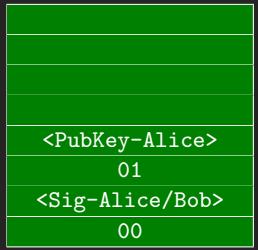
Time Lock Example — Second Case

```
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
```



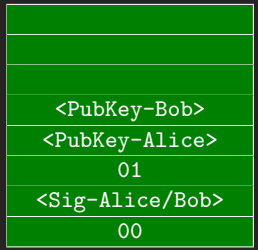
Time Lock Example — Second Case

```
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
```

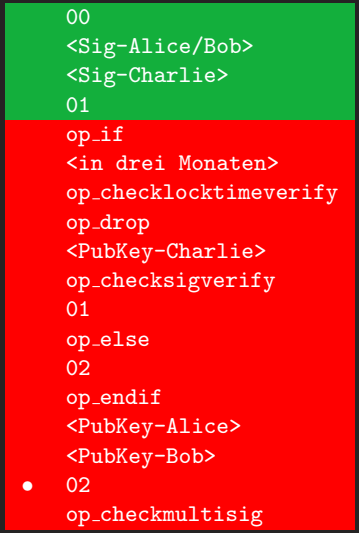


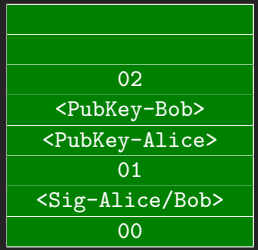
Time Lock Example — Second Case

```
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
```



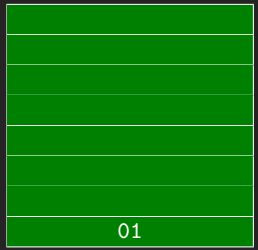
Time Lock Example — Second Case

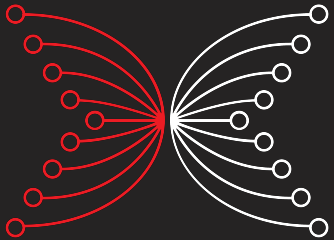




Time Lock Example — Second Case

```
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
```





INPUT | OUTPUT