



BLOC-521 Digital Currency Programming

How Bitcoin works, Part 1

Konstantinos Karasavvas



Learning objectives

- Introduce transactions and how they are propagated
- Explain how transactions become blocks and how mining works



Session outline

- The Story of a Transaction
- From Transactions to Blocks
- Mining
- Conclusions
- Further reading



Section 1: The Story of a Transaction



Transaction Basics (1)

Transactions specify the transfer of bitcoin ownership. Assume Alice has 1.5 bitcoins (BTCs) and wants to send 1 BTC* to Bob. The transaction history will already have an entry of how Alice got her bitcoins (e.g. from Zed as seen in TX_x).

At some point in the future Alice will create a transaction TX_y that sends 1 BTC to Bob. We know that Alice has at least 1.5 BTC from TX_x.

TX_x: 1Zed transfers 1.5 BTC to 1Alice
TX_y: 1Alice transfers 1 BTC to 1Bob

The names 1Zed, 1Alice and 1Bob are short for the actual bitcoin addresses of Zed, Alice and Bob respectively. So Alice will send 1BTC from her 1Alice bitcoin address to Bob to his 1Bob address.

Note that Alice has to prove that she is indeed the owner of the address 1Alice when she creates the TX. Bob does not need to do anything to receive the bitcoins.

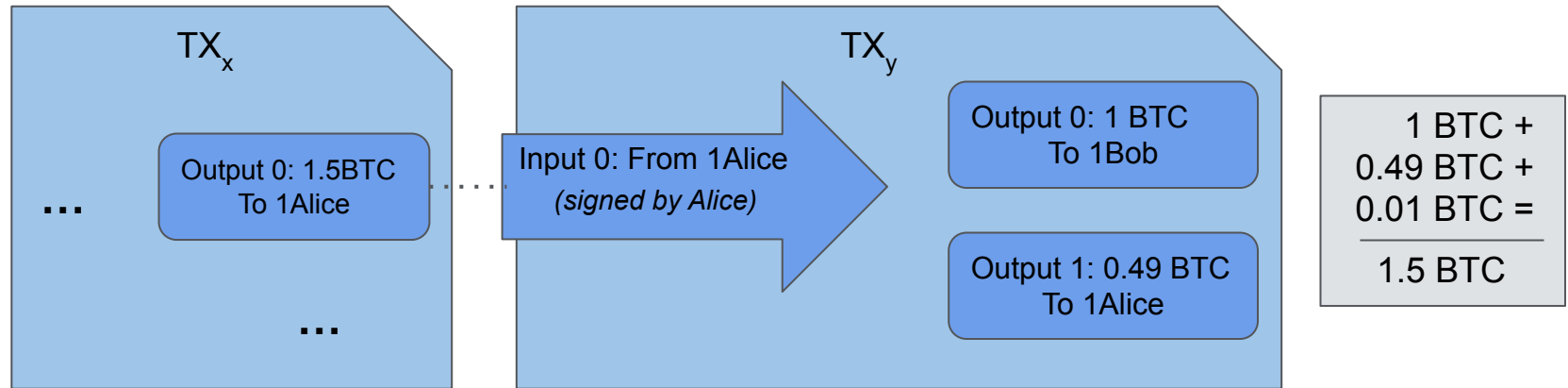
* Internally satoshis are sent. 1 satoshi = 0.00000001 BTC



Transaction Basics (2)

A transaction can consist of several **inputs** (addresses to get bitcoins from) and several **outputs** (addresses to send bitcoins to). When an input is used it is completely consumed; i.e. all the bitcoins that the TX contained need to be **spent**.

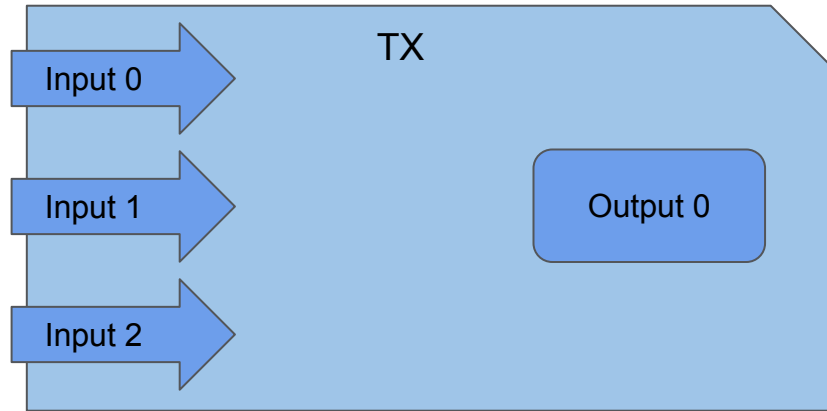
The amount of all the inputs needs to be greater or equal to the amounts of outputs. If greater (recommended) the difference is an implied *transaction fee* that goes to the miners.



Transaction Basics (3)

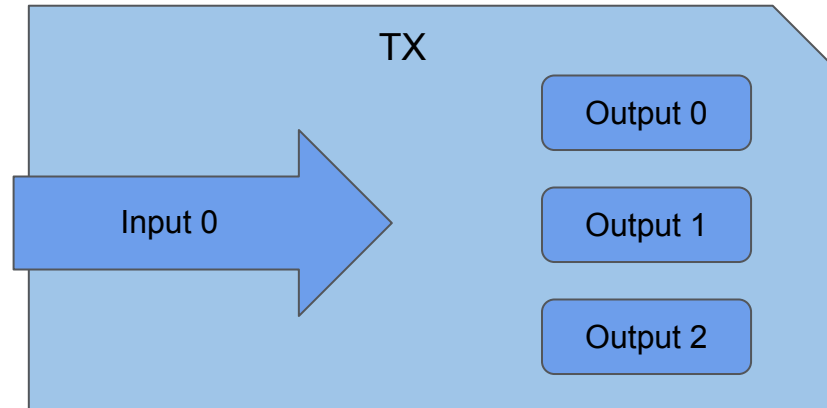
A typical transaction transfer some bitcoins to another user and returns the remaining bitcoins as *change* to the originating address or another address that the sender controls.

Another common transaction aggregates several inputs into a single output. This represents the real-world equivalent of exchanging a pile of coins and currency notes for a single larger note. Transactions like these are sometimes generated by wallet applications to cleanup lots of smaller amounts that were received as change for payments.



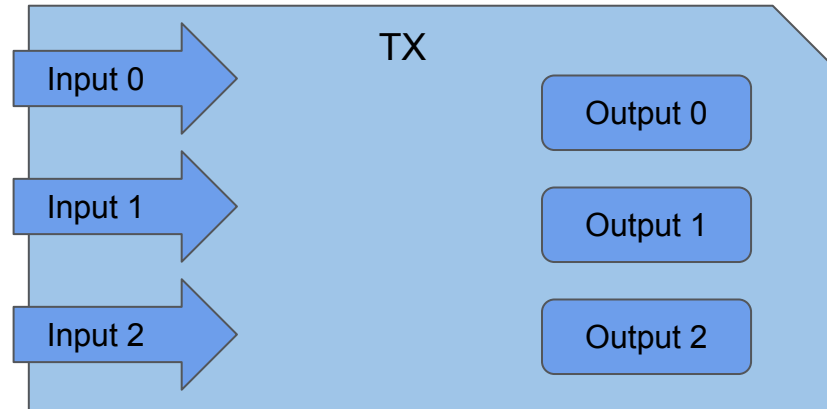
Transaction Basics (4)

Another transaction that is seen often on the Bitcoin network is a transaction that distributes one input to multiple outputs representing multiple recipients. This type of transaction is sometimes used by commercial entities to distribute funds, such as when processing payroll payments to multiple employees.



Transaction Basics (5)

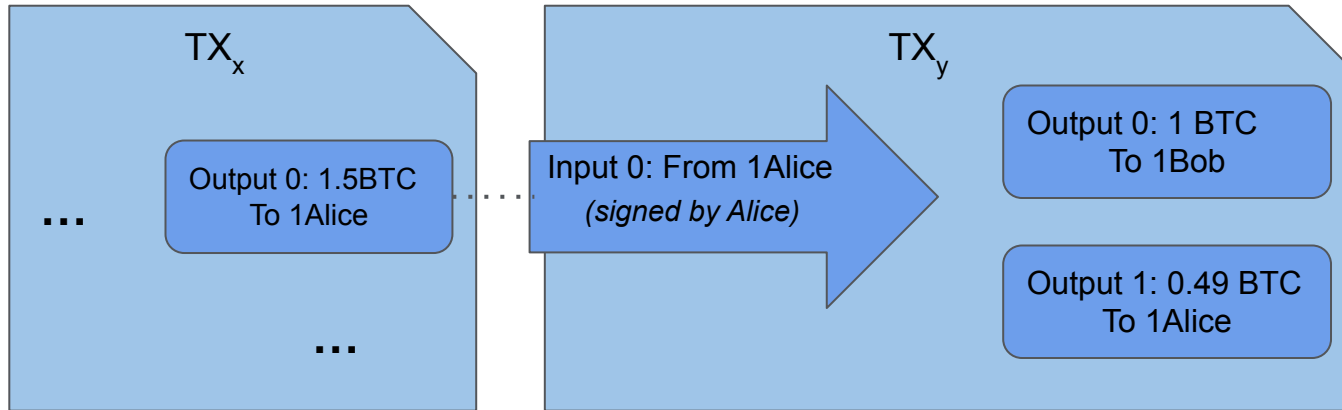
Finally any combination of inputs and outputs is possible.



Transaction Basics (6)

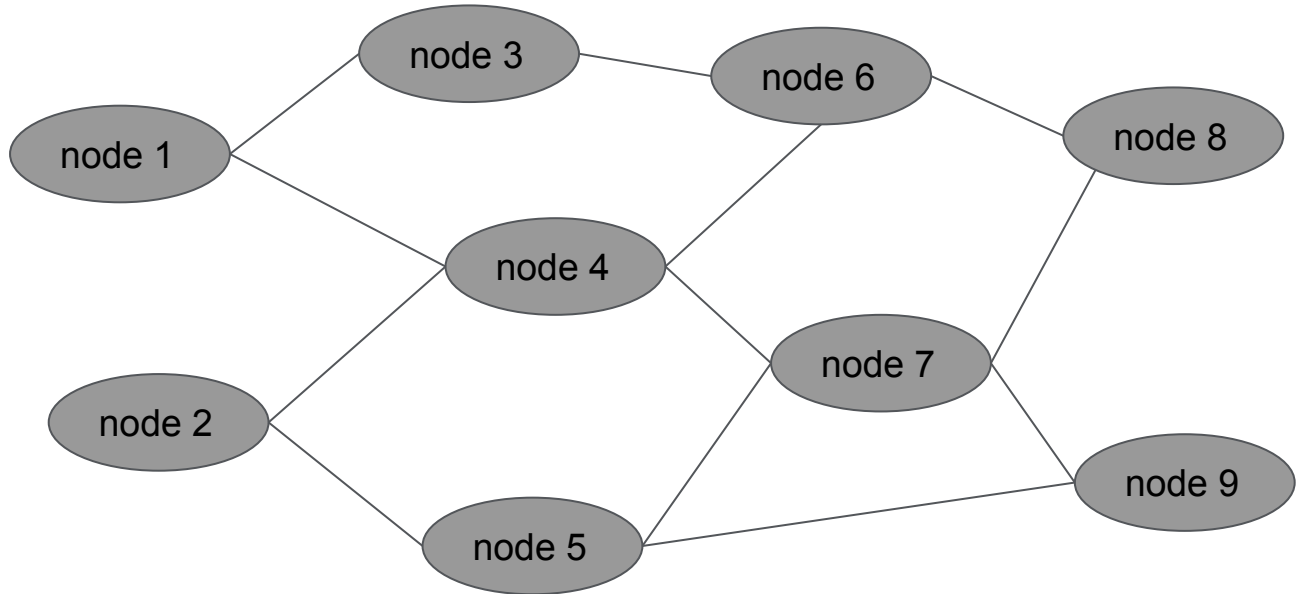
Note that outputs that haven't been spent yet are called Unspent Transaction Outputs (UTXO) and the set of UTXOs is essentially all the available bitcoins in the system.

So for our initial example Alice creates TX_y to send 1 BTC to Bob. What next?



Transaction Network Propagation

The transaction is sent to a bitcoin node.

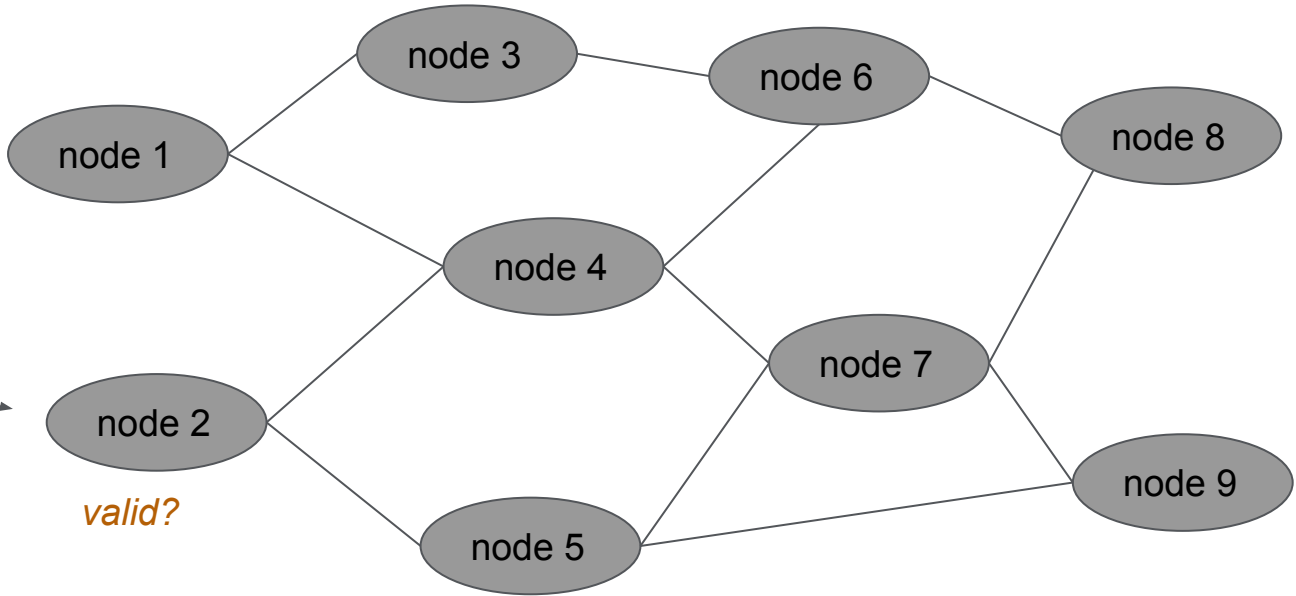


Transaction Network Propagation

The node checks if it is valid.



tx



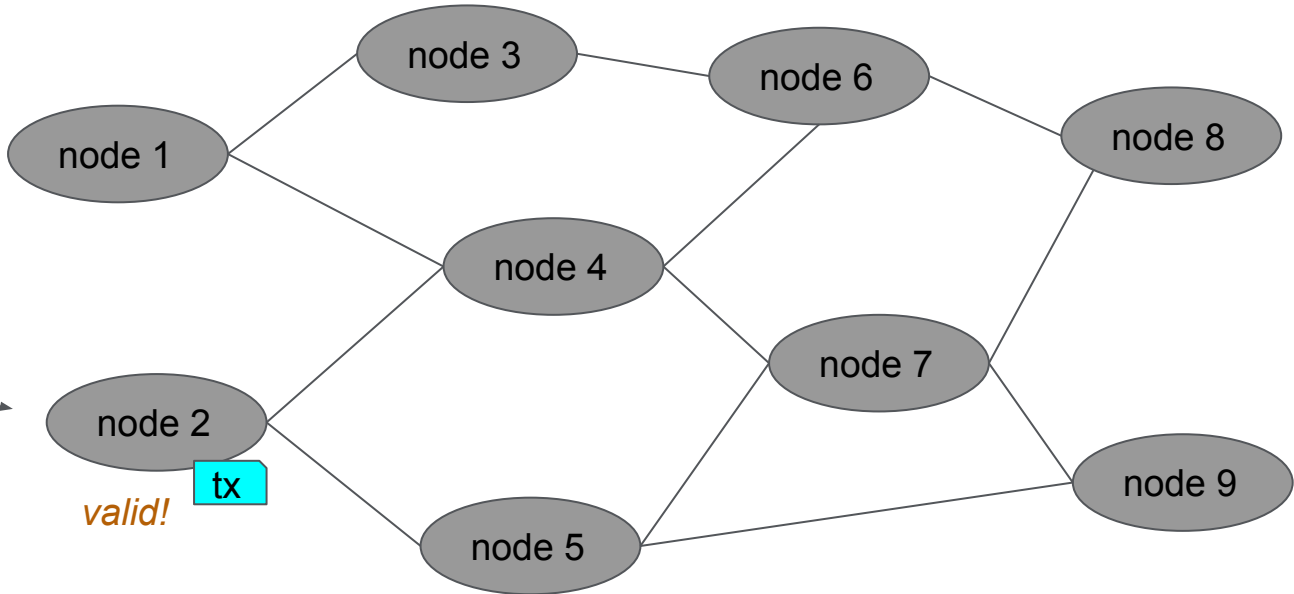
valid?

Transaction Network Propagation

And propagates to other nodes if it is.



tx

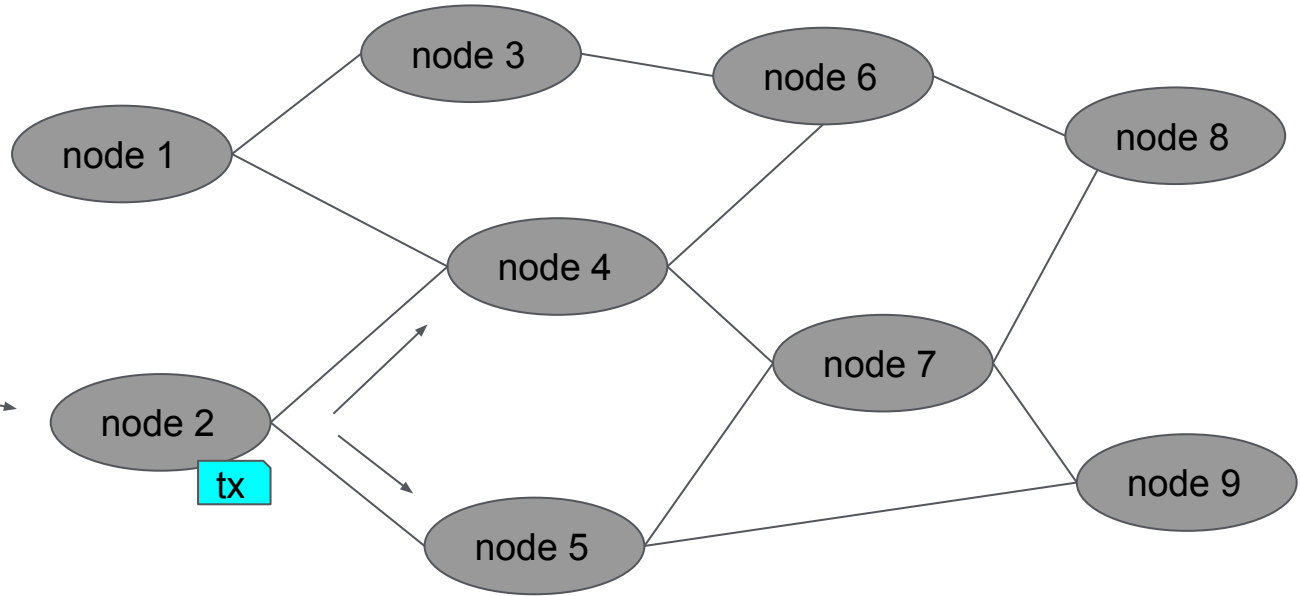


Transaction Network Propagation

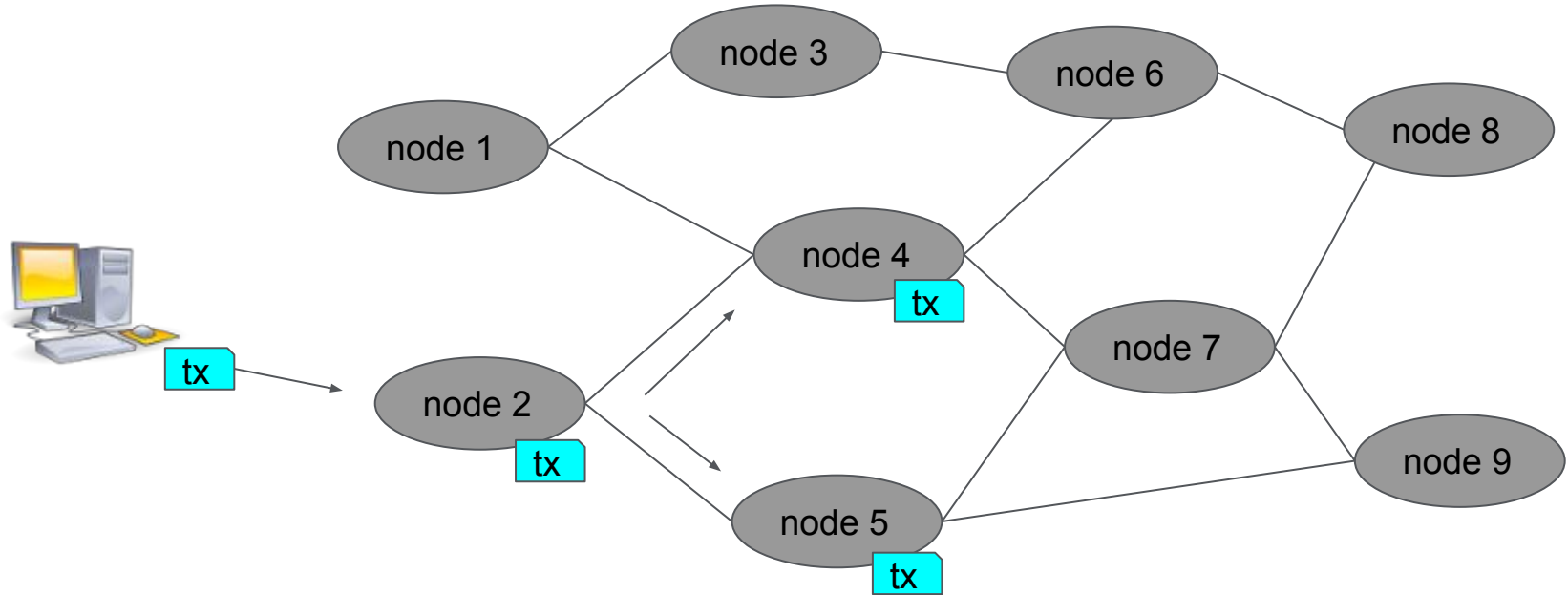
And so on and so forth until all nodes receive the transaction.



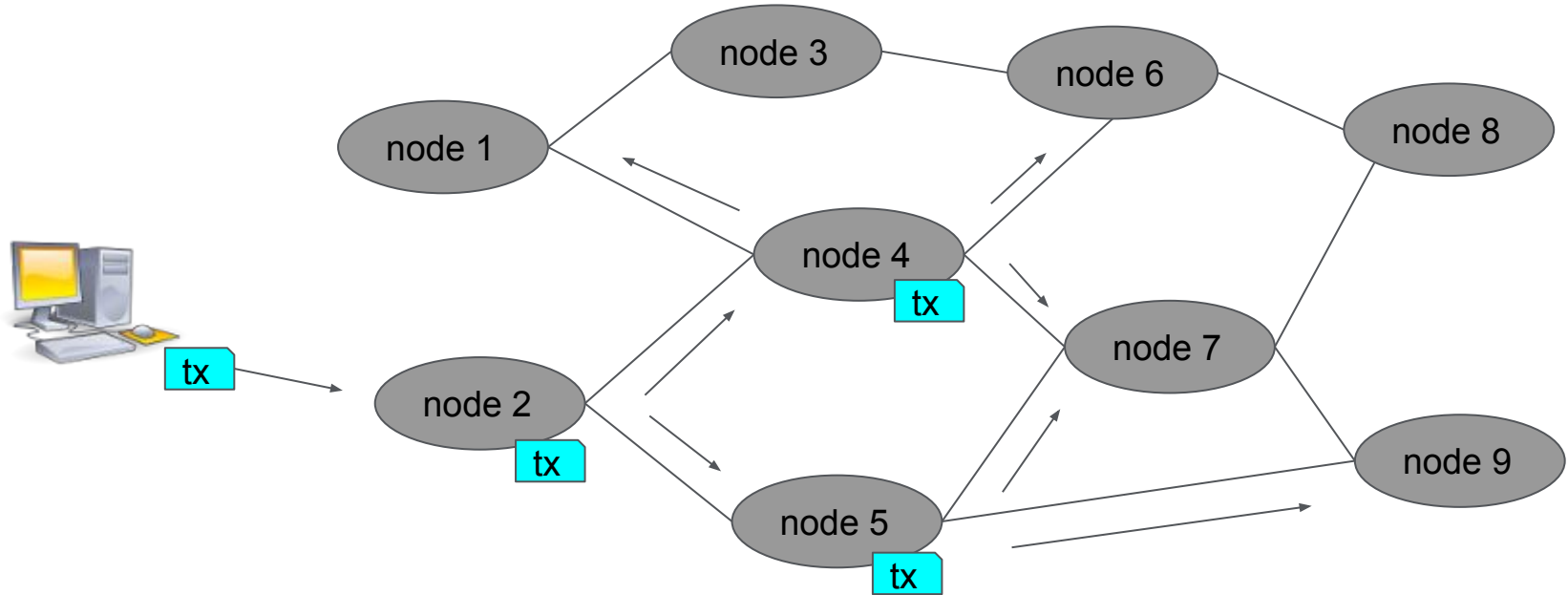
tx



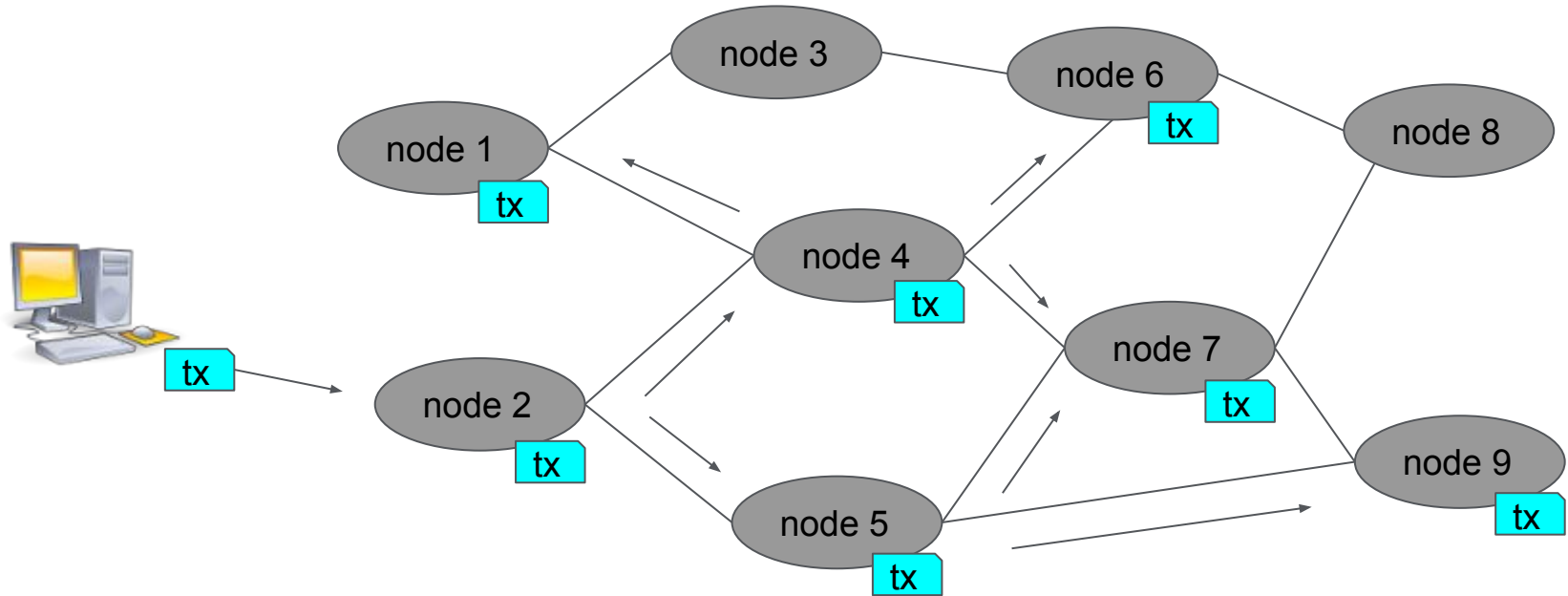
Transaction Network Propagation



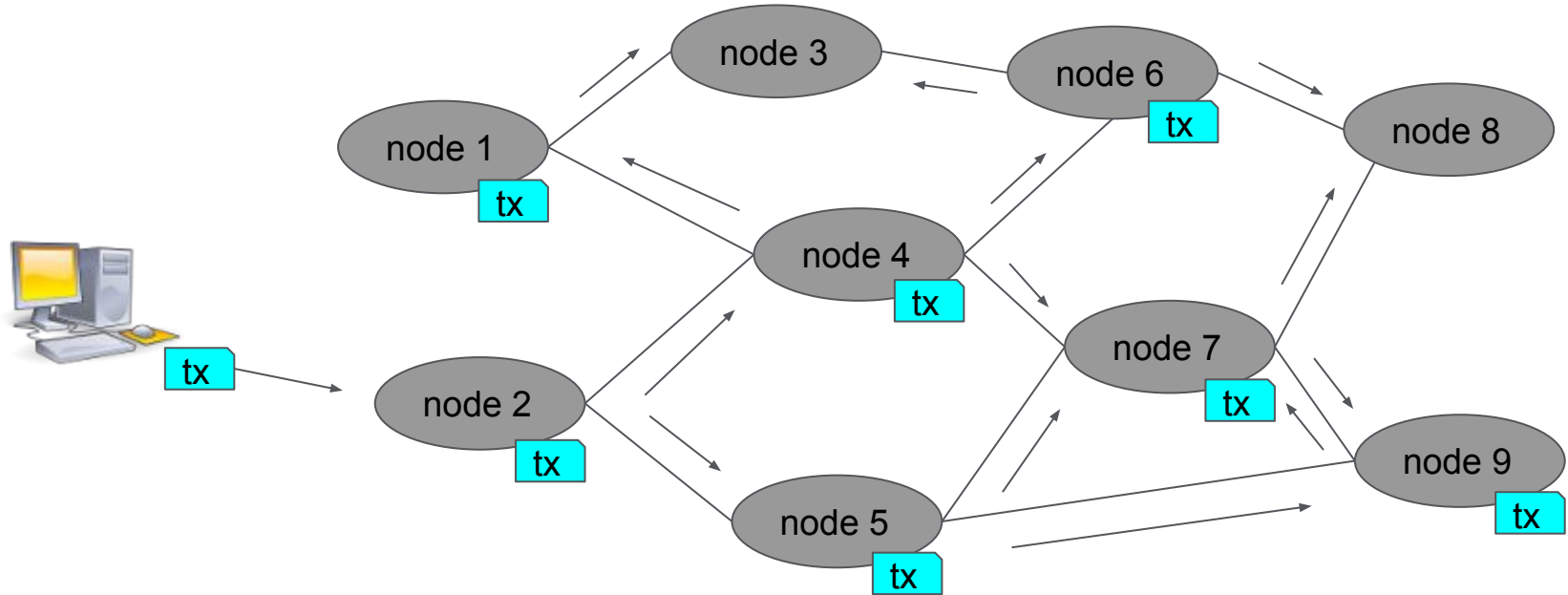
Transaction Network Propagation



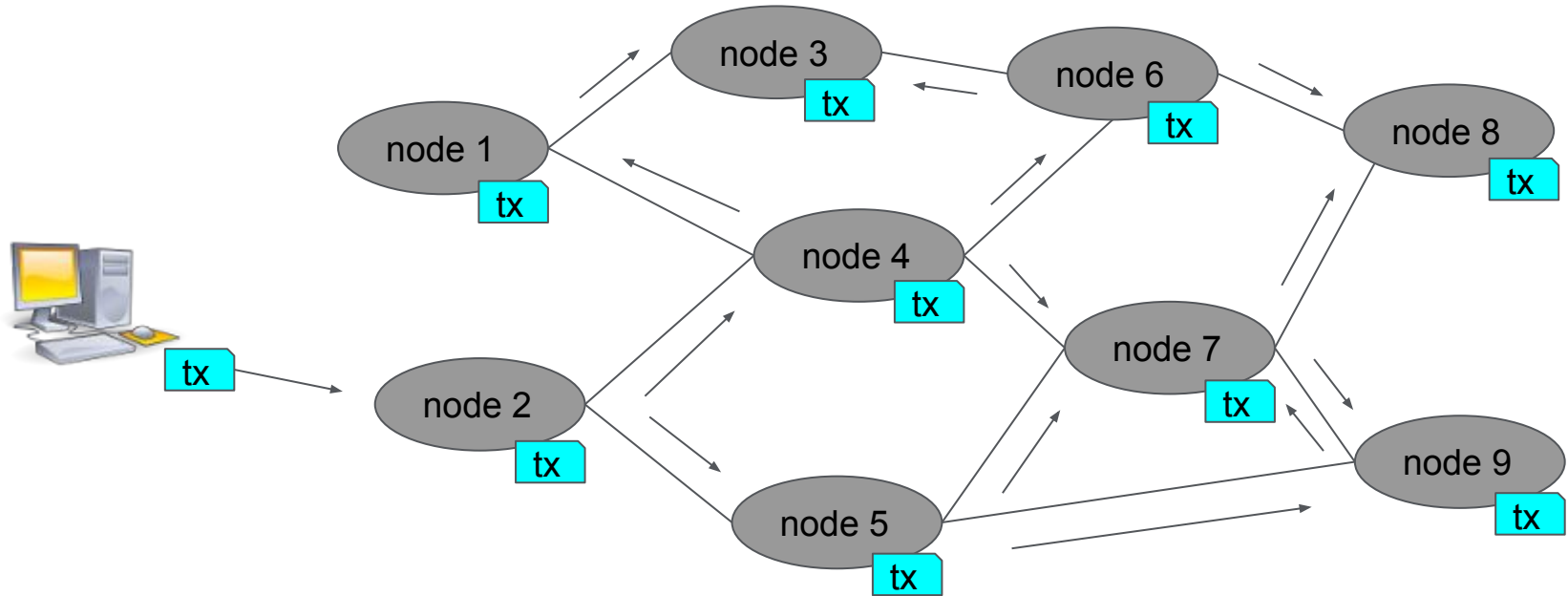
Transaction Network Propagation



Transaction Network Propagation



Transaction Network Propagation

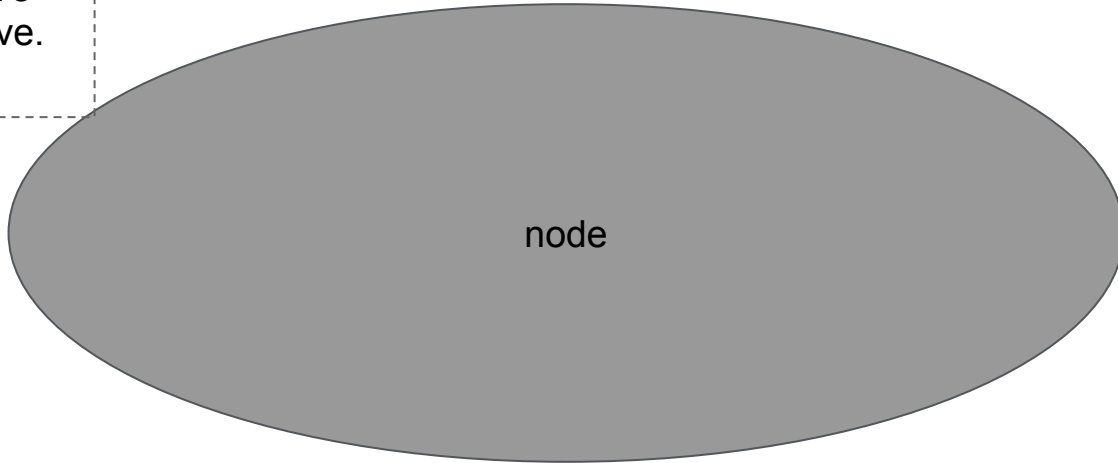


Section 2: From Transactions to Blocks



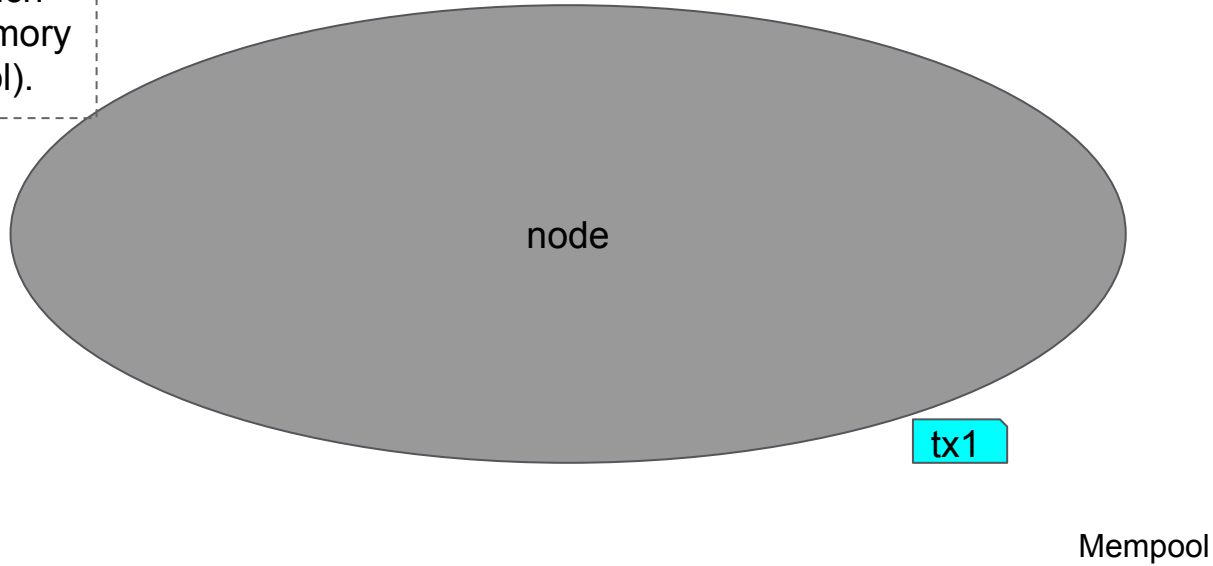
From Transactions to Blocks

From a Bitcoin's
node perspective.



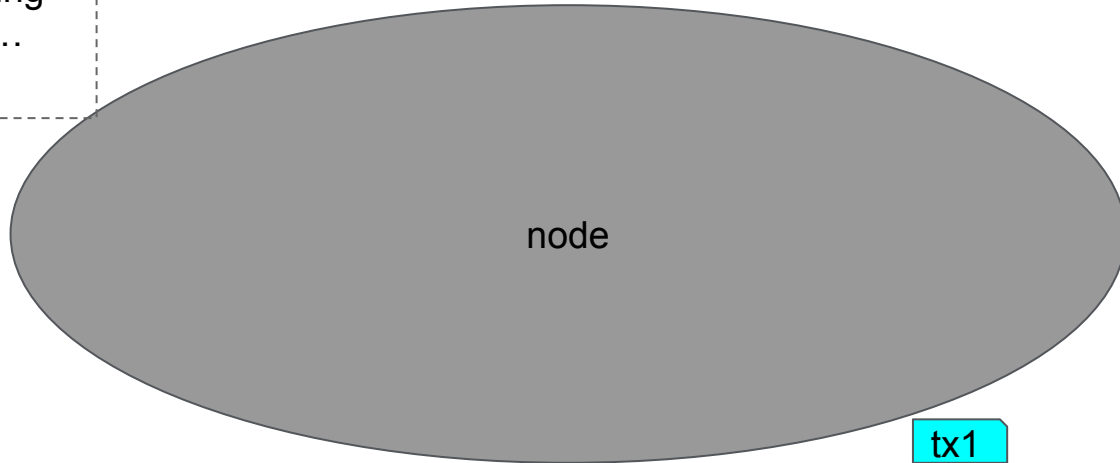
From Transactions to Blocks

The node receives a transaction which goes into its memory pool (mempool).



From Transactions to Blocks

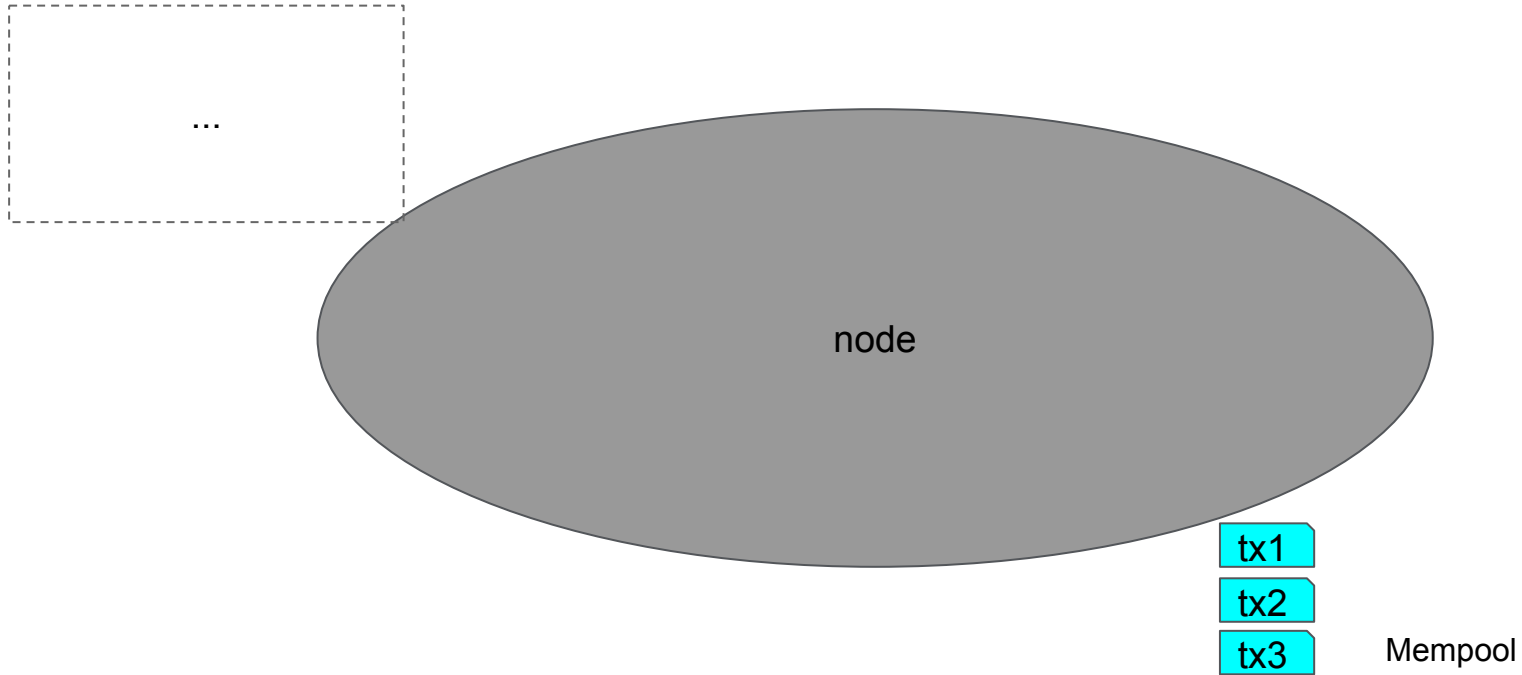
It keeps receiving transactions ...



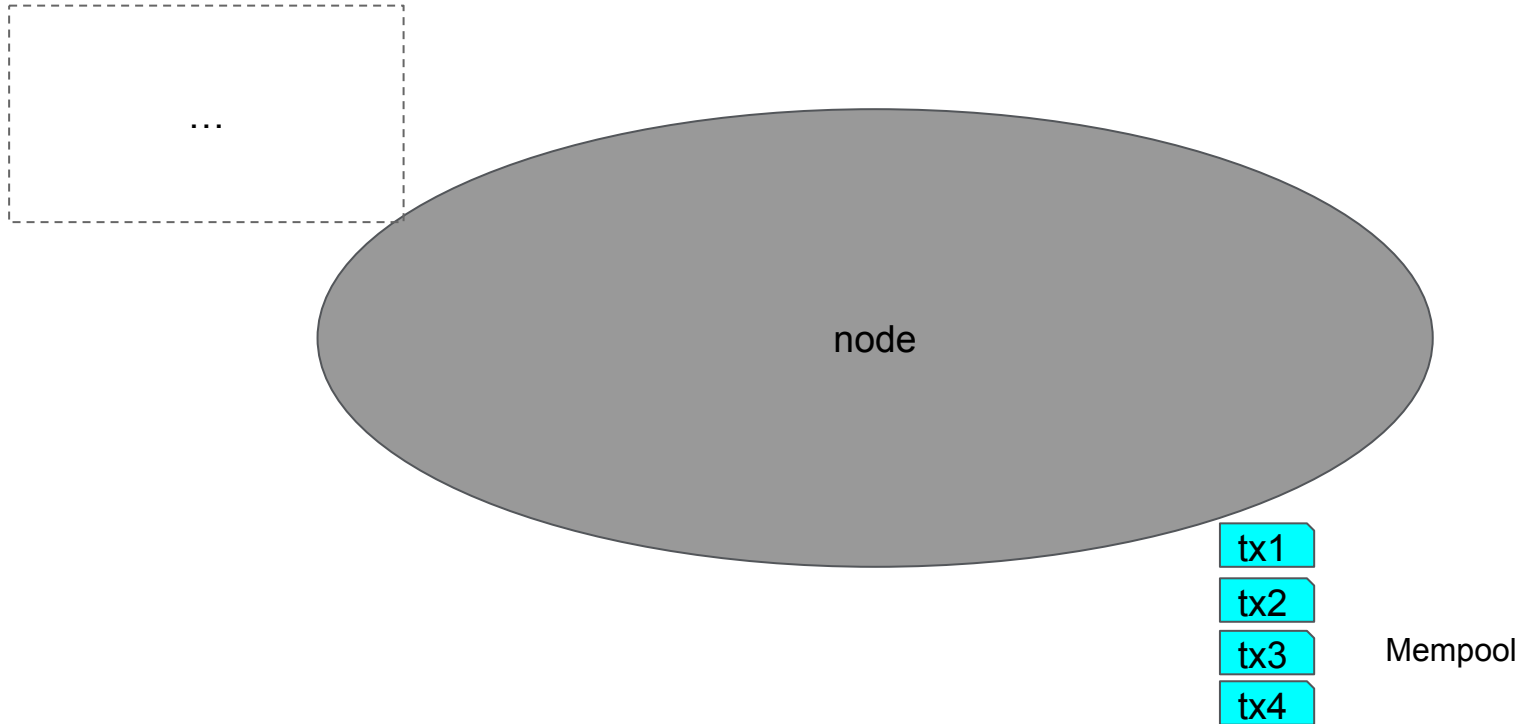
tx1
tx2

Mempool

From Transactions to Blocks

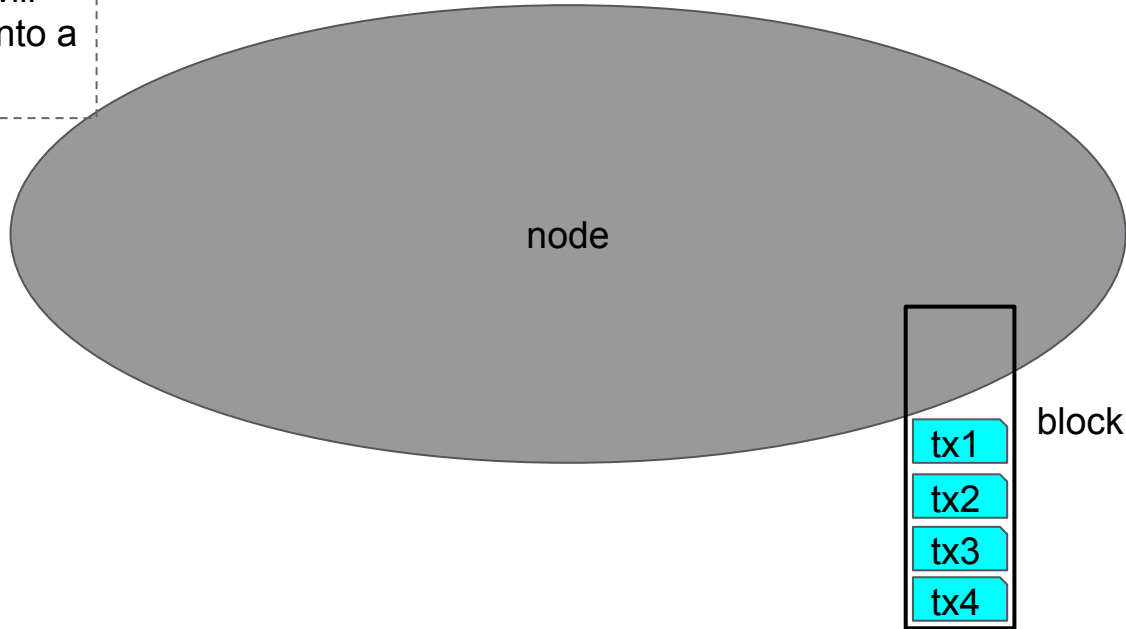


From Transactions to Blocks



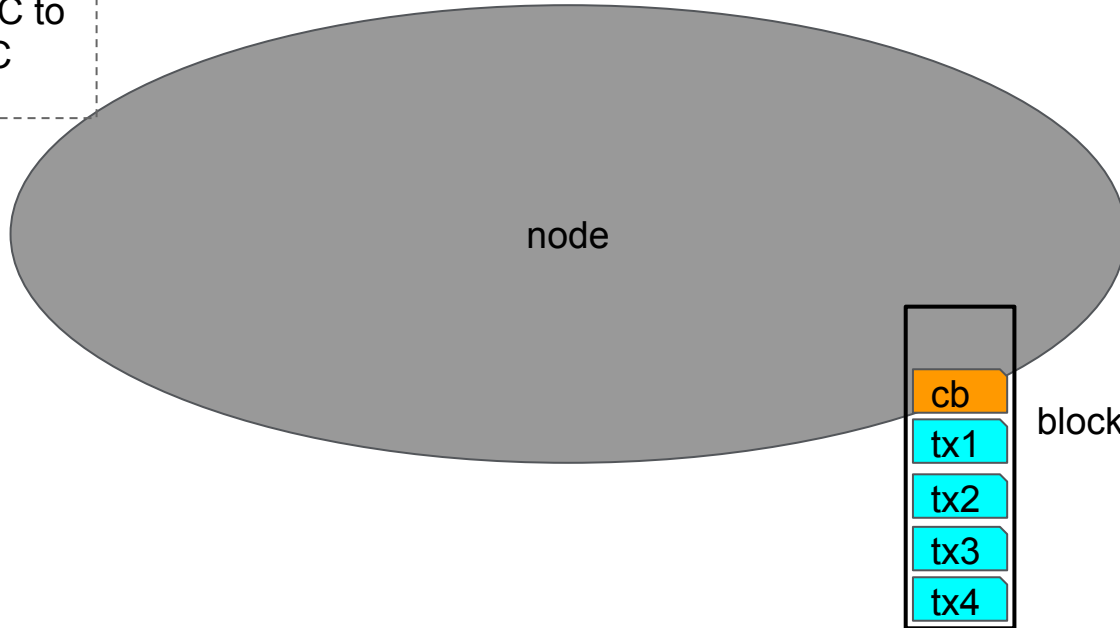
From Transactions to Blocks

At some point it will decide that it will group these txs into a block.



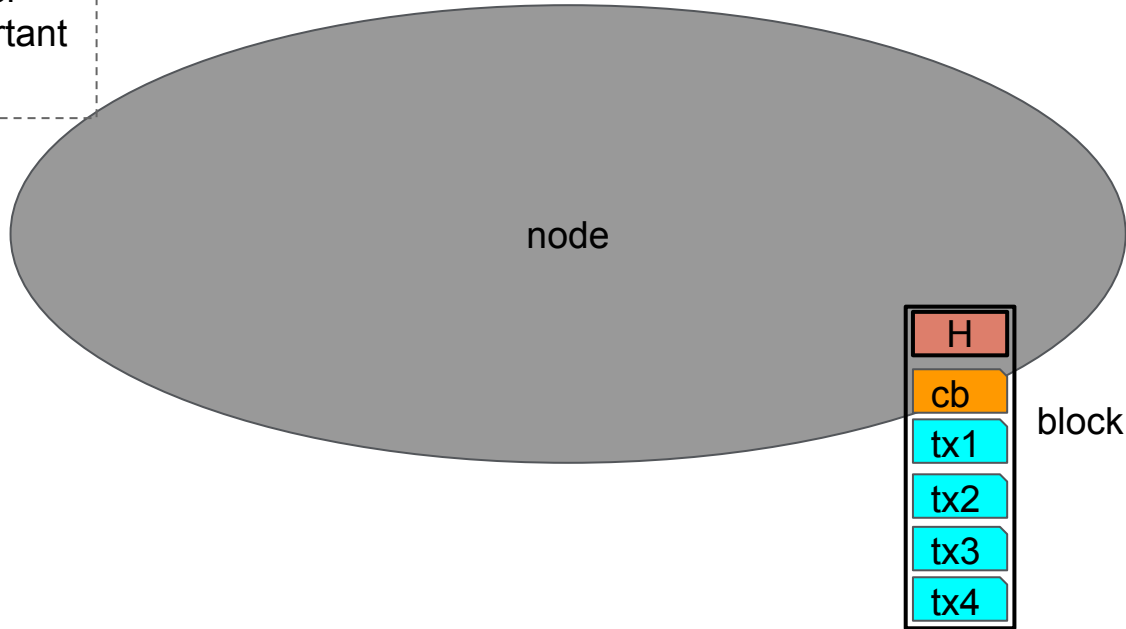
From Transactions to Blocks

It will add the coinbase tx which rewards 6.25BTC to a node's BTC address.



From Transactions to Blocks

Finally it adds the block's header containing important information.



Section 3: Mining



Mining a Block (1)

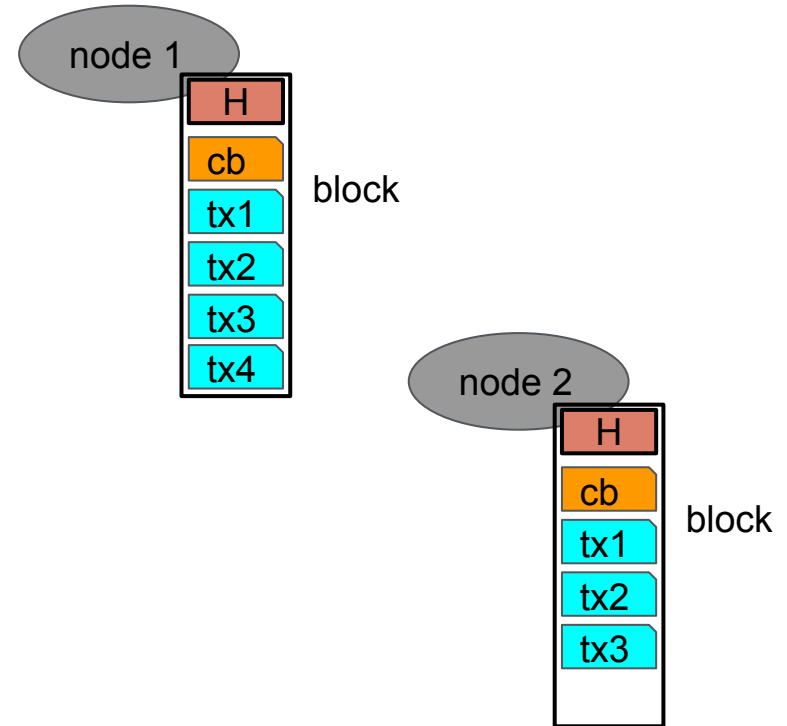
After a node creates a block it will attempt to make it final by propagating it to all other nodes in the network.

Multiple nodes will receive the same transactions and will create blocks; nodes choose which TXs to include. They can create and propagate a block at any time.

How do we avoid spam?

Which blocks are accepted by the network?

A very difficult computational problem needs to be solved in order to accept a block as valid. The process of finding the solution requires work (Proof-of-Work) and is called mining. The mining problem has the fundamental property of being difficult to calculate but trivial to validate its correctness.

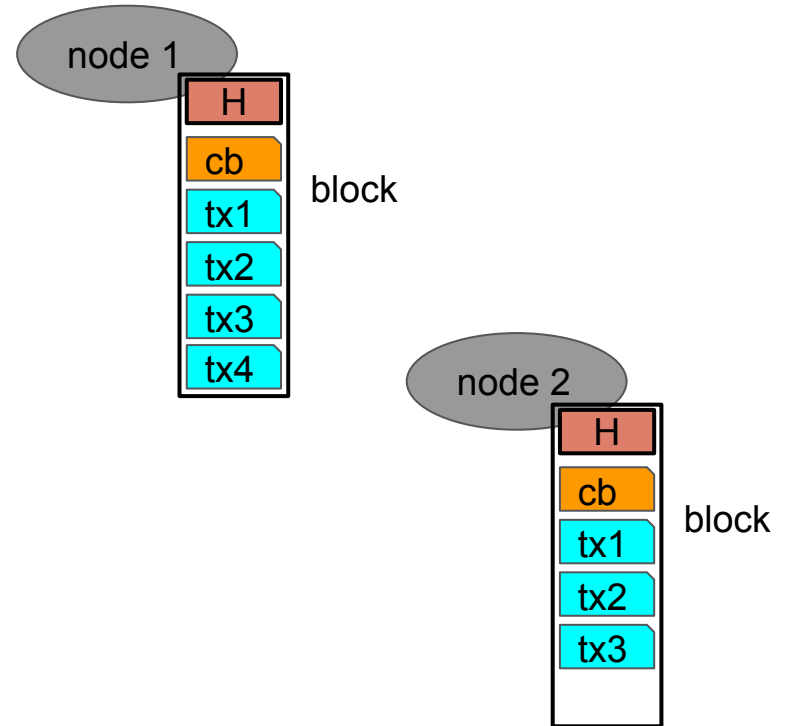


Mining a Block (2)

The Proof-of-Work puzzle is to compute a cryptographic hash* of the new block that we want to create which should be less than a given number. Since a hash is random it will take several attempts to find a proper hash but other nodes will verify with only one attempt.

$\text{SHA256}(\text{SHA256}(\text{block_header}))$

* A cryptographic hash function is a hash function that takes an arbitrary block of data and returns a fixed-size bit string, the cryptographic *hash value*, such that any (accidental or intentional) change to the data will also change the hash value significantly.

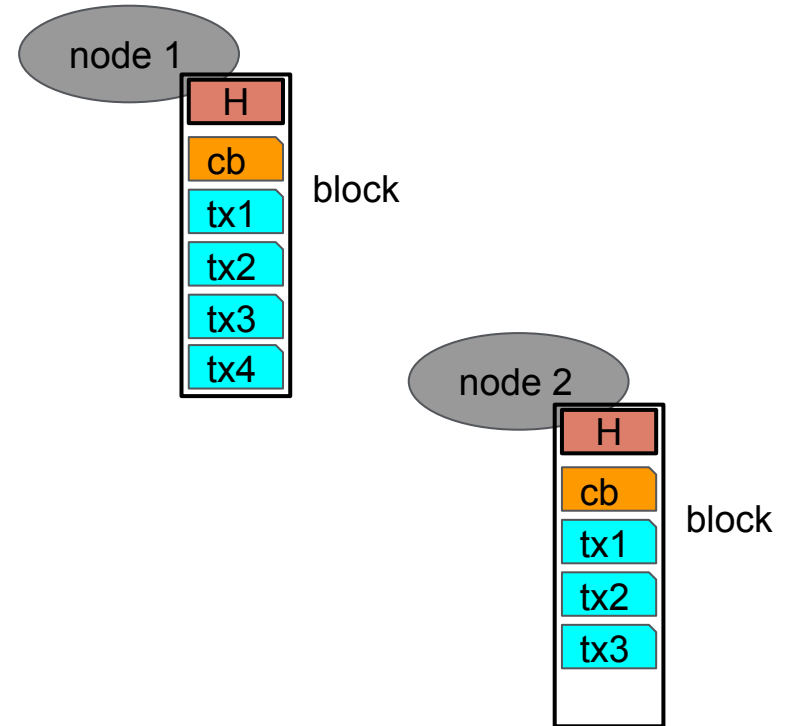


Mining a Block (3)

The puzzle's difficulty automatically adjusts so that it requires approximately 10 minutes to solve. As more miners join the blocks will be created faster. The difficulty of the puzzle is then increased to require ~10 minutes again

This *difficulty adjustment* is happening every 2016 blocks (approximately 2 weeks if each block takes 10 minutes to mine).

The coinbase transaction is added by the miner and is a reward of 6.25 BTC (the current agreed-upon reward) to an address that the miner controls. I.e. if his block is accepted he will get the reward.



Mining a Block (4)

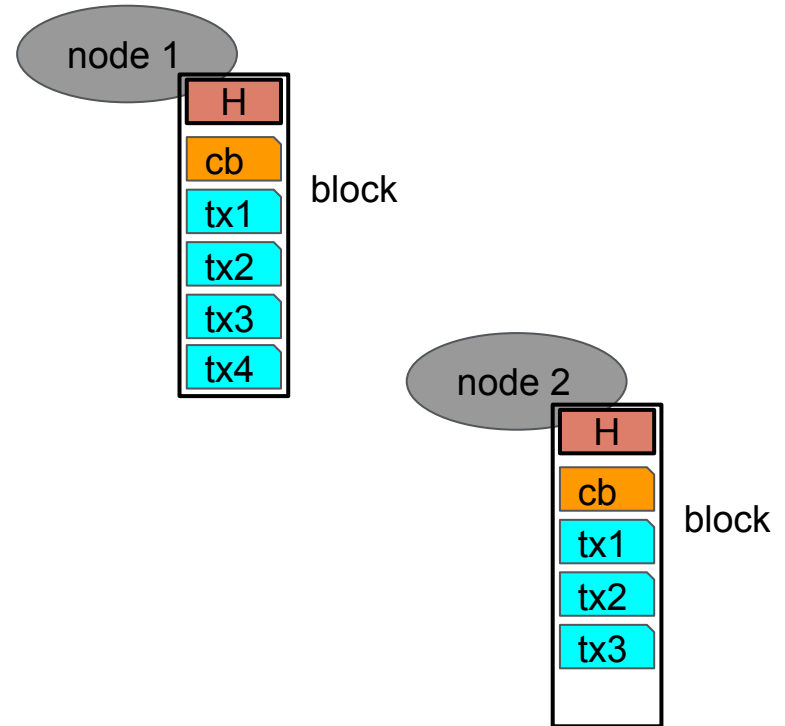
The reward started at 50 bitcoins and is halved every 210000 blocks (approximately 4 years).

Additionally, the transaction fees of all the TXs in a block are also awarded to the miner that creates the new block.

The header of a block contains, among other things, a link to the previously created block .

A block always contains a coinbase transaction which is used to pay the mining reward to the miner.

The mining reward is available to the miner after 100 confirmations.



Block Header

Field	Description	Size (bytes)
version	Block version number	4
hashPrevBlock	256-bit hash of the previous block	32
hashMerkleRoot	256-bit hash representing all the TXs in the block	32
timestamp	Seconds since 1970-01-01T00:00 UTC	4
target (bits)	the target that the hash should be less than	4
nonce	32-bit number	4

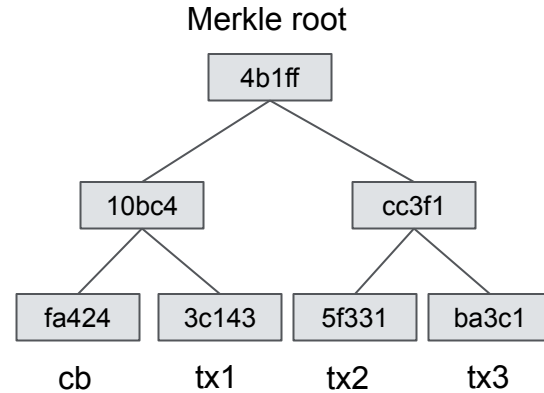


hashMerkleRoot

Since we only hash the block header to link blocks together, a header needs to represent the whole block, including all its transactions (coinbase and normal). The transactions are indirectly hashed via using a merkle root.

A merkle tree is constructed by concatenating all the transaction hashes, in pairs. The resulting hashes are again concatenated and hashed until only a single hash remains, the merkle root.

A merkle proof consists of the hashes and their position in order to reach from a leaf TX hash to the merkle root, thus proving that that TX hash is indeed part of the merkle tree.



Target (1)

Target bits or just bits is represented as an 8 hex-digit number. The first 2 digits are the exponent and the rest the coefficient.

Target bits can be used to calculate the actual target with the following formula:

$$\text{target} = \text{coefficient} * 2^{(8 * (\text{exponent} - 3))}$$

The resulting target would be a 64 hexadecimal number (256-bits), e.g.

```
0x00000000ffff000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

If hashing the block header produces a hash that begins with `0x00000000e` (or less) then we have found a solution. That would require, statistically $\sim 2^{32}$ (4,294,967,296) attempts on average. The smaller the target the more difficult the solution, the more attempts on average.



Target (2)

```
$ ./bitcoin-cli getbestblockhash
00000000000000000149df9c7bd55689c0d34fd55361a430d8858f1fc559105c
```

That would require 2^{68} (295,147,910,000,000,000) attempts on average.

The highest possible target (easiest target, *difficulty* 1) is defined as `0x1d00ffff` and gives a hex target of: `0x00000000ffff000`

Another representation of target, easier for humans to understand, is *difficulty* which represents the ratio between the highest target and the current target ($D = \text{max} / \text{current}$).

The easiest target (difficulty 1) requires $\sim 2^{32}$ attempts thus a difficulty of 10 requires $2^{32} * 10$ attempts.

nonce

The nonce is just a number used to differentiate the hash while trying to reach the target. Given that it is only 4 bytes it can only handle $\sim 4.2B$ combinations, while we need quadrillions nowadays.

When the limit was reached miners started modifying the timestamp (e.g. -1 sec) to allow for an additional of $\sim 4.2B$ combinations. However, there is a limit of seconds that a node can deviate from the rest of the network so that did not suffice either.

Then miners started to use the coinbase transaction as an extra nonce allowing an immense amount of extra nonces to be used.



Mining Process in a nutshell

- Gather valid TXs into blocks
- Get the longest chain's top block hash and add it in hashPrevBlock
- Add timestamp, nonce and extra nonce in the first TX (coinbase)
- Calculate the merkle root of valid TXs and add it to hashMerkleRoot
- Hash the header to find a solution smaller than the specified target
 - modify timestamp, nonce or extra nonce as appropriate
 - rehash until a solution is found or the longest chain changed
- Meanwhile:
- If more TXs are included in the block or the extra nonce is modified
 - recalculate merkle root and update it
- If the longest chain changed we want to build on that chain from now on
 - update the valid TX set
 - recalculate the merkle root
 - use the new block as hashPrevBlock



Difficulty adjustment

The difficulty to find the proper hash is expected to take approximately 10 minutes. However, Bitcoin is an open system and anyone can join (or leave) the network as a miner. Thus, the network's hashrate can increase (or decrease) with time.

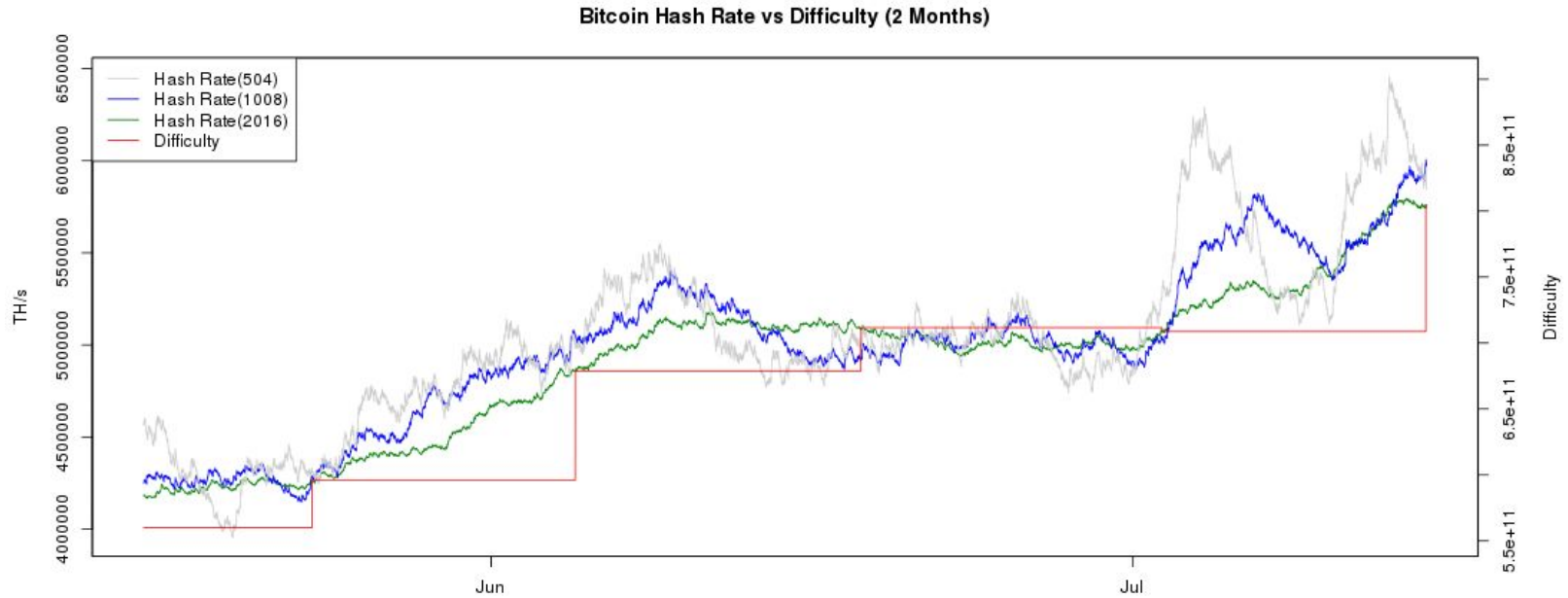
With more hashing power blocks will be issued faster than 10 minutes and thus the network has to adjust the difficulty of the problem accordingly.

Specifically, Bitcoin nodes, check every 2016 blocks (~2 weeks) the timestamps between consecutive blocks and sums them to find out how much time t it took. We want t to take two weeks and thus the new difficulty will be:

$$\text{difficulty} * (2 \text{ weeks} / t)$$



Hash Rate vs Difficulty (2017-07-14)



Conclusions



Conclusions

- We went through the process of how a transaction is created in the Bitcoin network, the basic structure of a transaction and common types.
- We explained how transactions are propagated between nodes and how blocks are created from transactions.
- We have seen how a block is mined and the process of mining.



Further reading



Self-assessment exercises

- Prepare a bitcoin environment by installing a Bitcoin node configured for testnet.
- Using bitcoin-cli create a new address
 - use a testnet faucet to get some test coins
- Using bitcoin-cli to send some test bitcoins to some of your classmates
 - share your testnet addresses via the forums
- Backup your wallet
- Go through the rest of the API and get familiar with more commands

You are welcome to use the forums to report issues, questions or your thoughts in general!



Further Reading

Bitcoin Programming Textbook (Ch.1), Kostas Karasavvas

<https://kkarasavvas.com/assets/bitcoin-textbook.pdf>

Mastering Bitcoin (Ch.2, Ch.10), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch02.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch10.asciidoc>

Mining Difficulty and how it is calculated

<https://en.bitcoin.it/wiki/Difficulty>





UNIVERSITY *of*
NICOSIA

The background is a dark blue, blurred image of a computer screen. It shows faint, glowing lines of code and data, suggesting a digital or programming environment. The text is overlaid on this background.

BLOC-521 Digital Currency Programming

How Bitcoin works, Part 2

Konstantinos Karasavvas



Learning objectives

- Show block propagation and how blocks are confirmed
- Explain Nakamoto Consensus
- Introduce the Bitcoin software and basic interactions with a node



Session outline

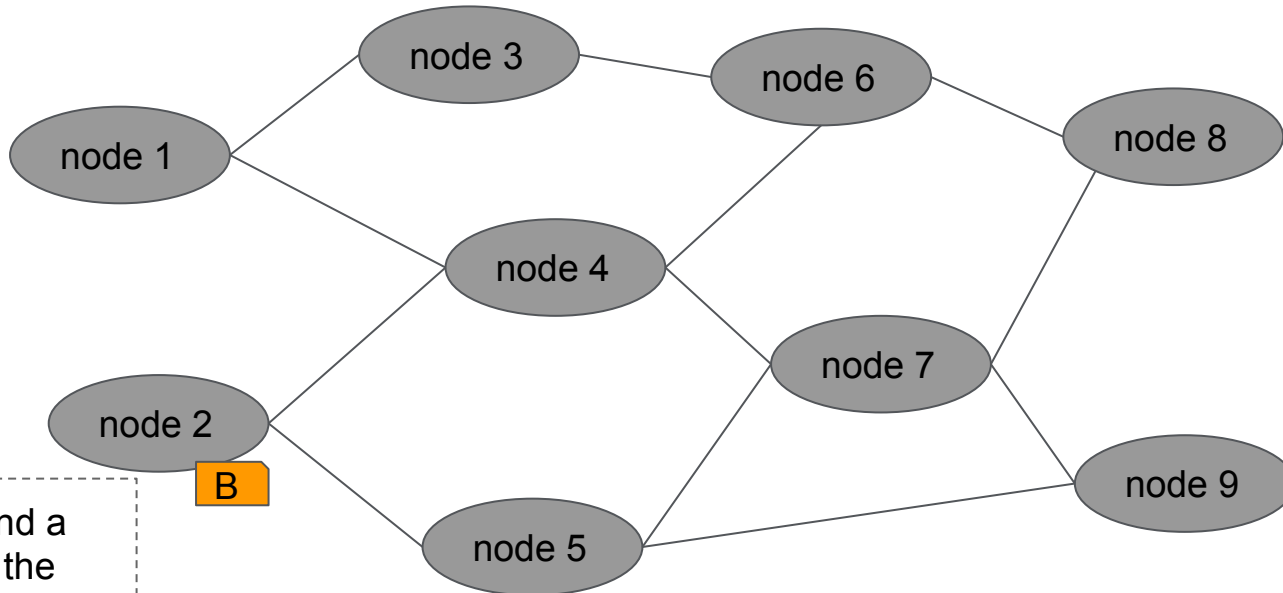
- The Story of a Block
- Nakamoto Consensus
- Basic interaction with a node
- Conclusions
- Self-assessment exercises and further reading



Section 4: The Story of a Block



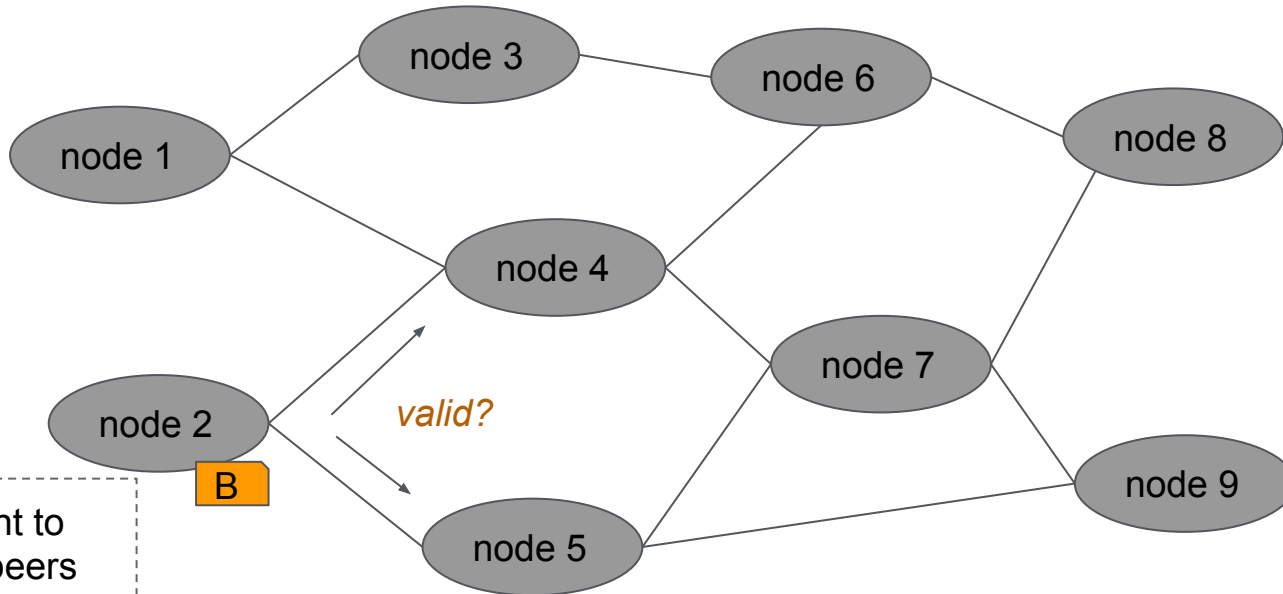
Block Network Propagation



Node 2 found a solution to the puzzle! Let's propagate!



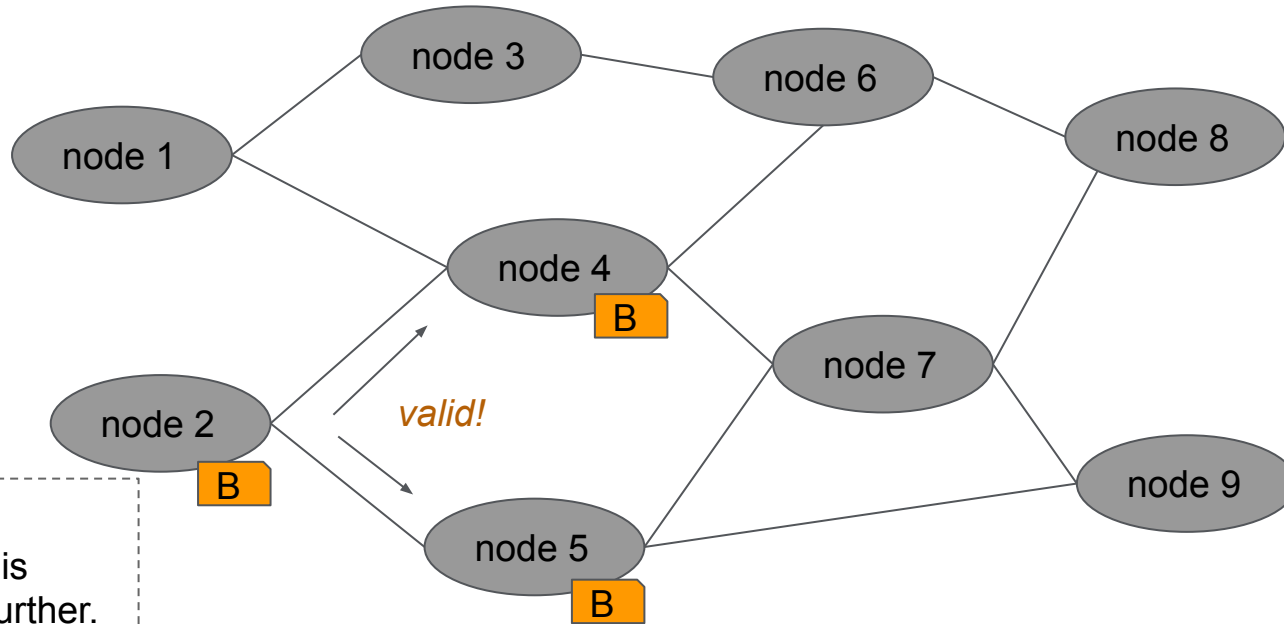
Block Network Propagation



Block is sent to connected peers where it is checked for validity.

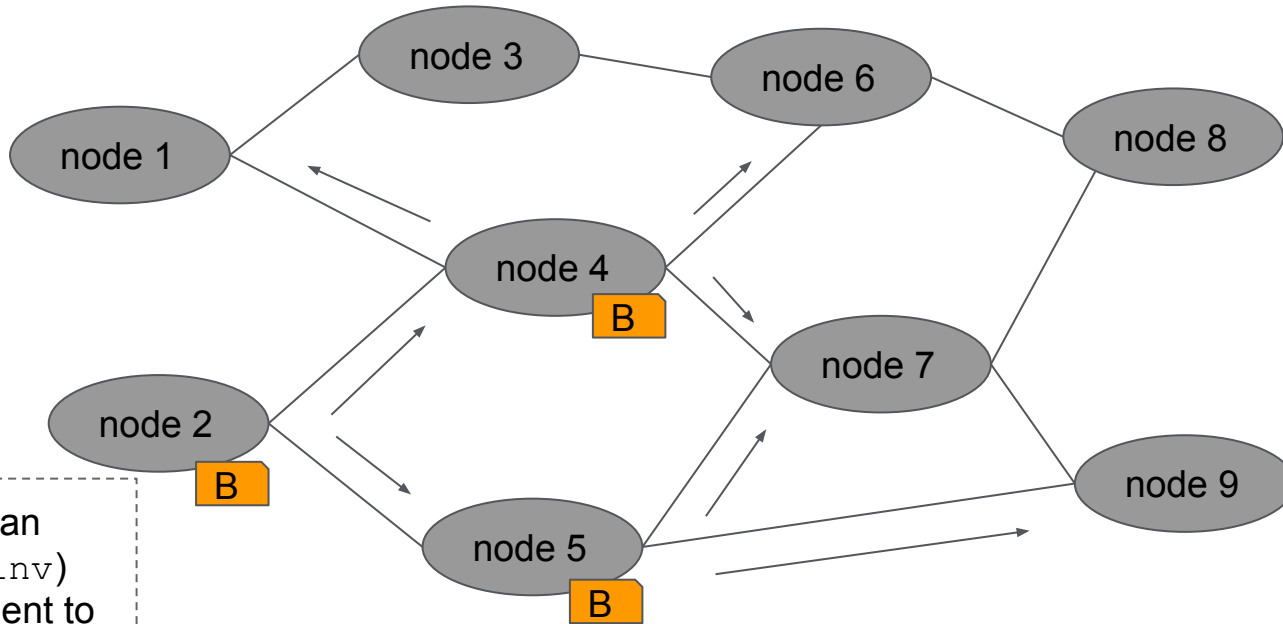


Block Network Propagation



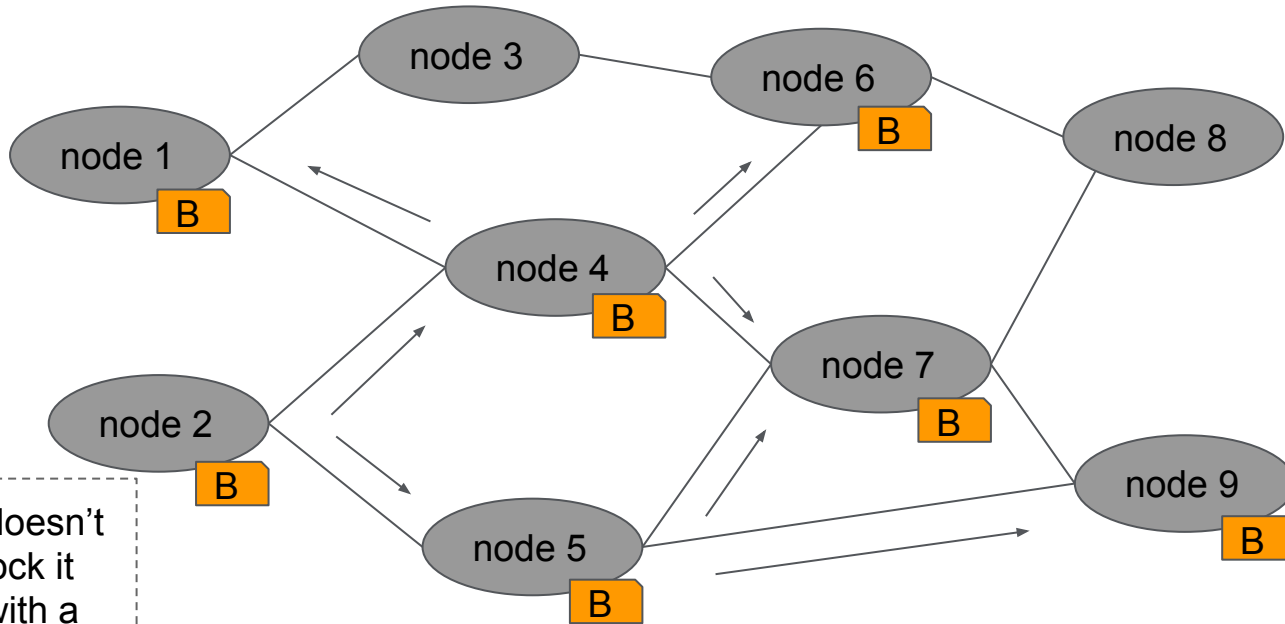
If valid it is propagated further.

Block Network Propagation



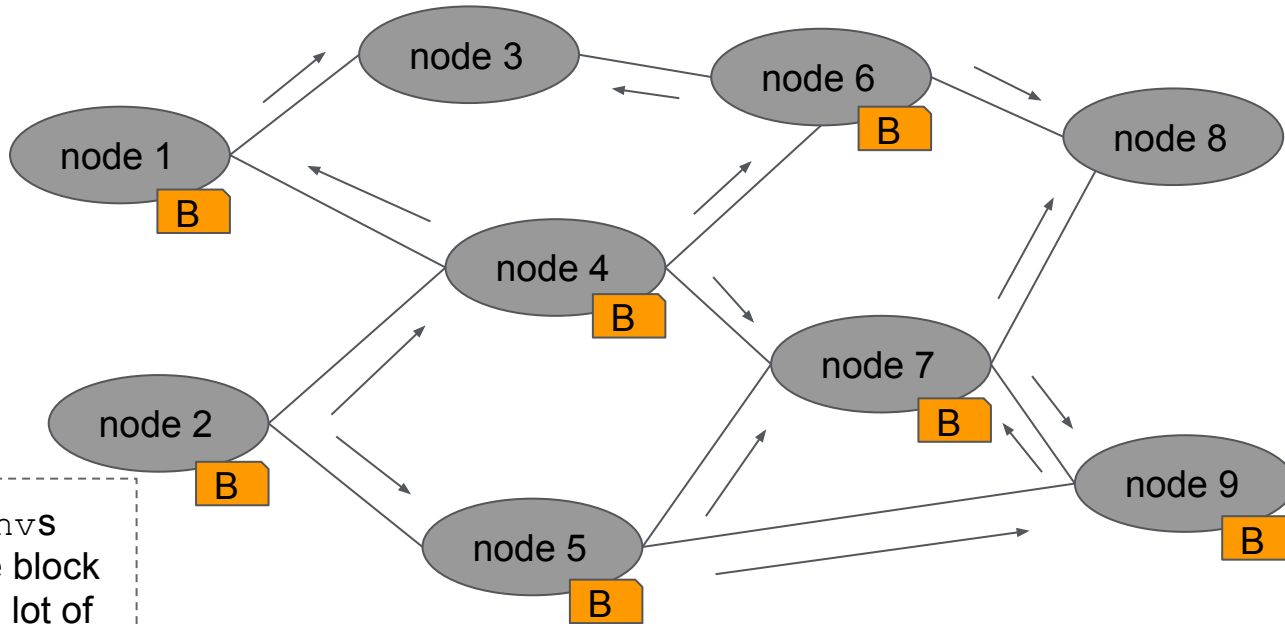
Note that an inventory (`inv`) message is sent to notify of a new block.

Block Network Propagation



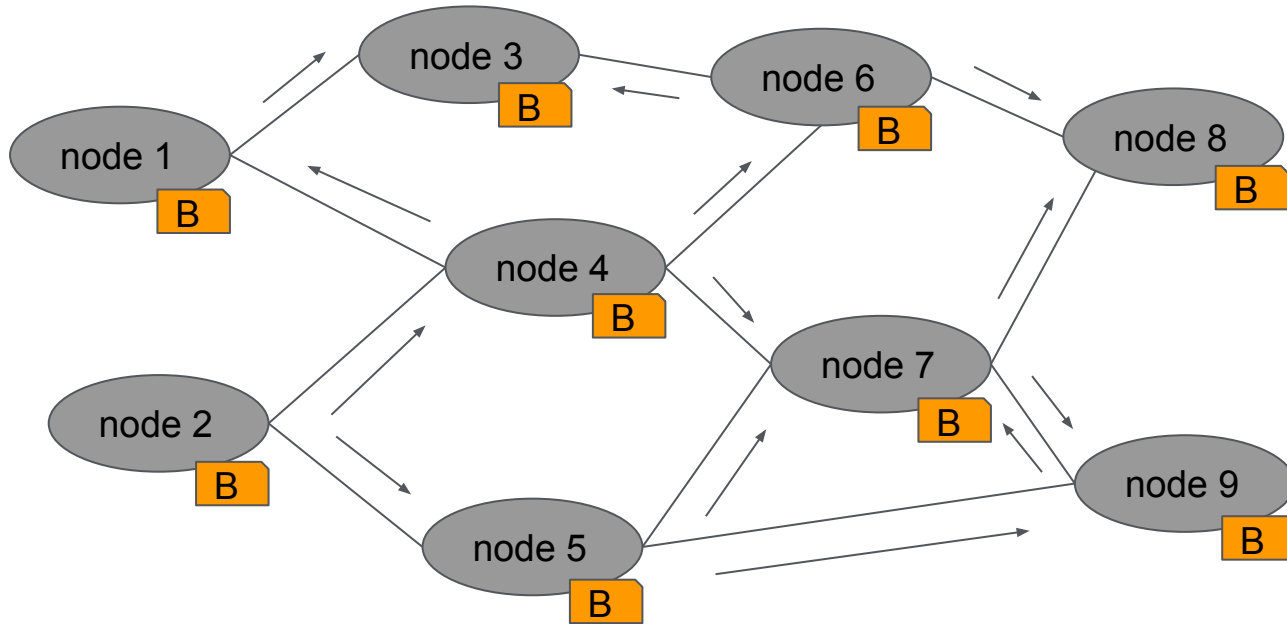
If peer node doesn't have the block it requests it with a `getdata` message.

Block Network Propagation



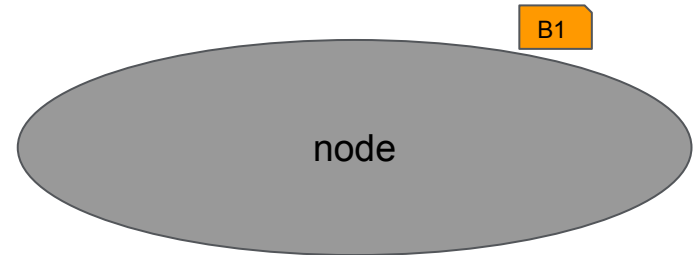
Sending `invs` instead of the block itself saves a lot of bandwidth.

Block Network Propagation



Forming a chain of Blocks

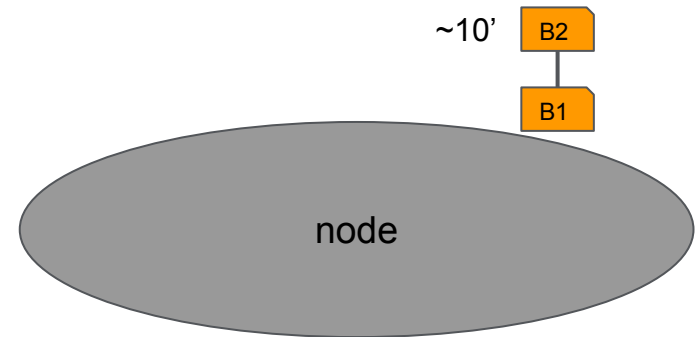
The new block is being added on top of the existing blocks (every ~10 minutes). This occurs on every single node on the network thus the blocks are the same in all nodes.



Forming a chain of Blocks

The new block is being added on top of the existing blocks (every ~10 minutes). This occurs on every single node on the network thus the blocks are the same in all nodes.

Blocks are linked with cryptographic hashes forming a chain of blocks, called *Blockchain*.



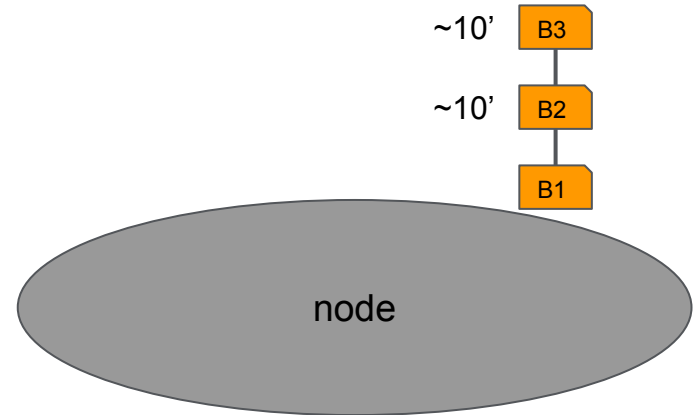
Forming a chain of Blocks

The new block is being added on top of the existing blocks (every ~10 minutes). This occurs on every single node on the network thus the blocks are the same in all nodes.

Blocks are linked with cryptographic hashes forming a chain of blocks, called *Blockchain*.

When Block B1 is accepted by the network we say that a transaction on that block has one confirmation. When B3 is accepted we say that our transaction has 3 confirmations.

The more confirmations the more final and secure a transaction is (detailed later in Blockchain and Trust).



Section 5: Nakamoto Consensus

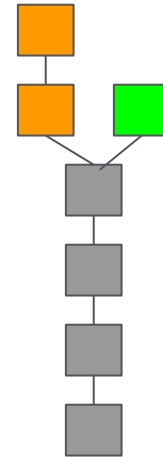


Nakamoto Consensus (1)

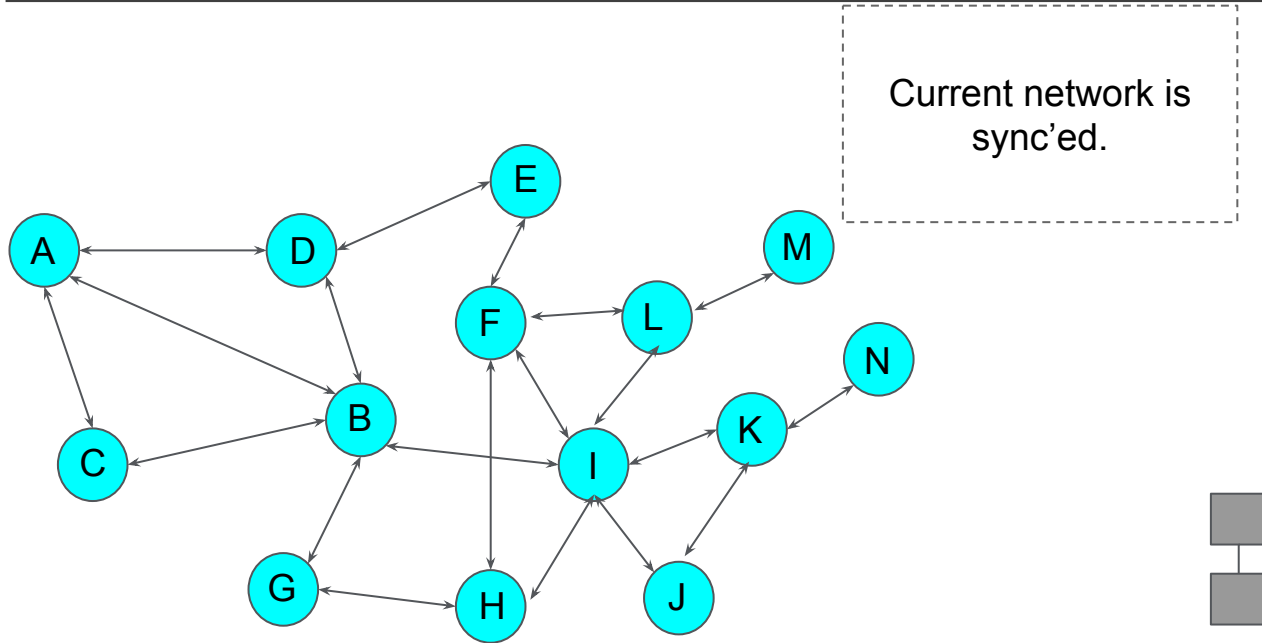
Each node receives blocks and builds its own blockchain in isolation. A fundamental innovation that bitcoin introduced is the Nakamoto consensus, i.e. how do different nodes come to agreement on what is the current state of the blockchain.

If two miners find a block (almost) at the same time then network peers will get a different block first. They will then start building the next block based on the one they received first. That means that the network at that time has two possible states.

In Nakamoto consensus the basic rule is that miners should **follow the longer chain** (the one with the most computation). Thus, when one of the miners finds the next block all miners will choose the longer chain and consensus is achieved.

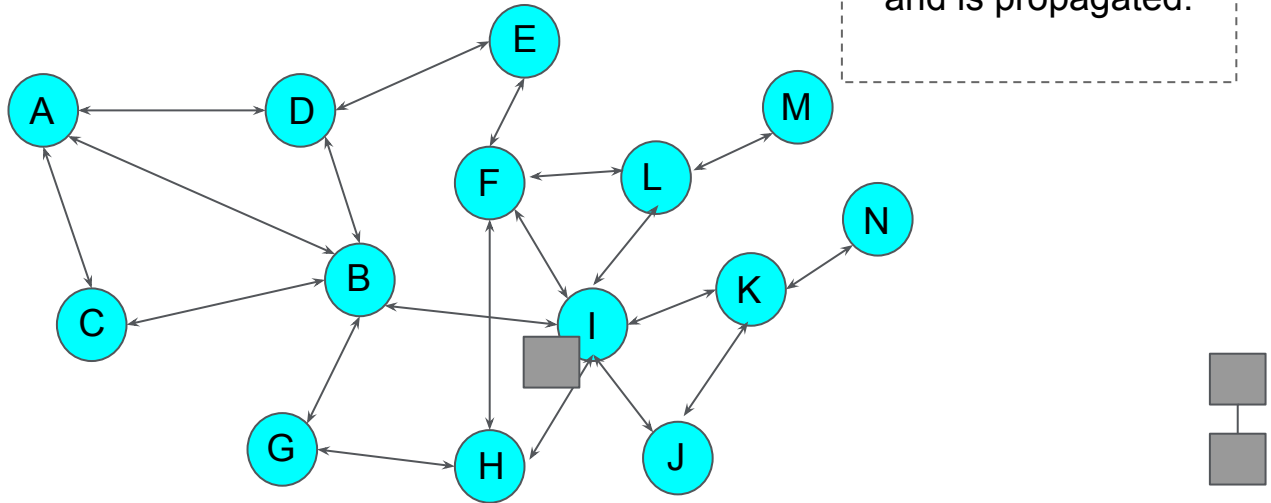


Nakamoto Consensus (2)

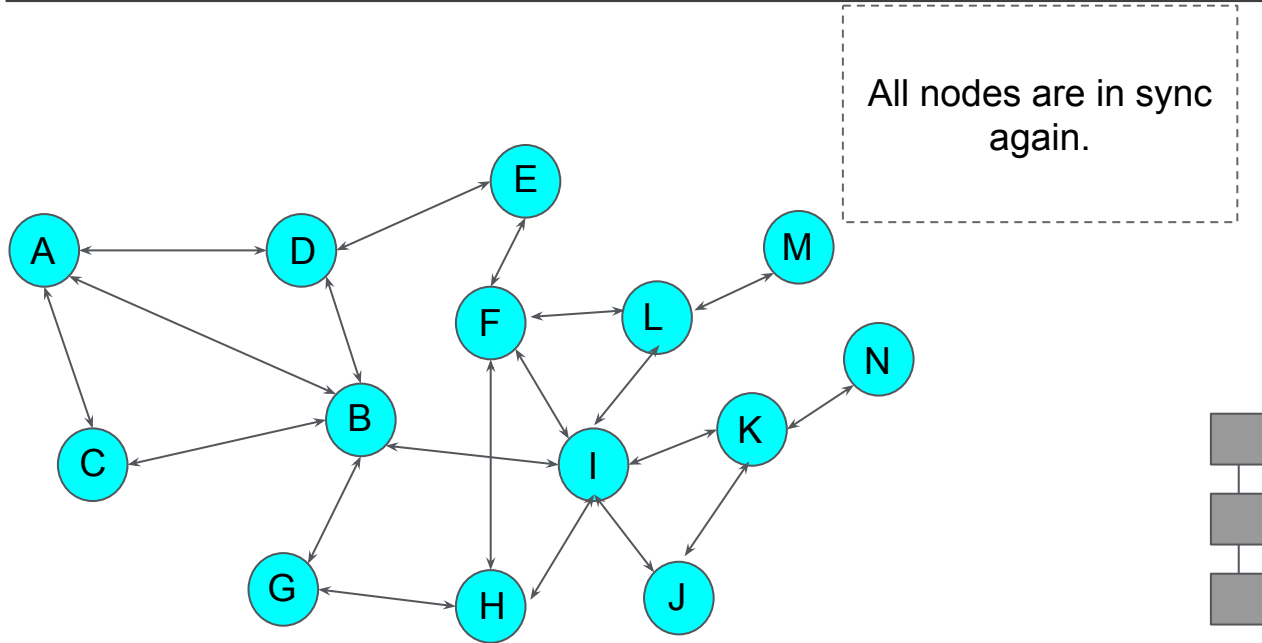


Nakamoto Consensus (2)

A new block is found and is propagated.

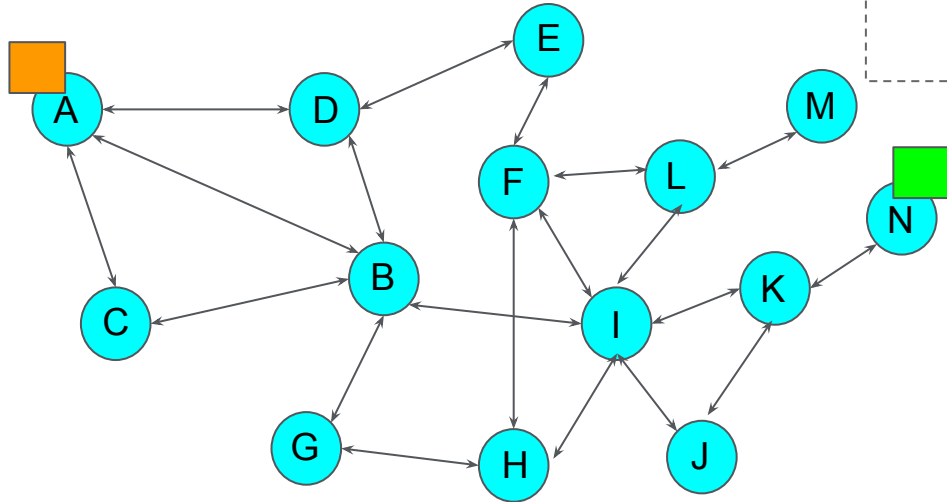


Nakamoto Consensus (2)

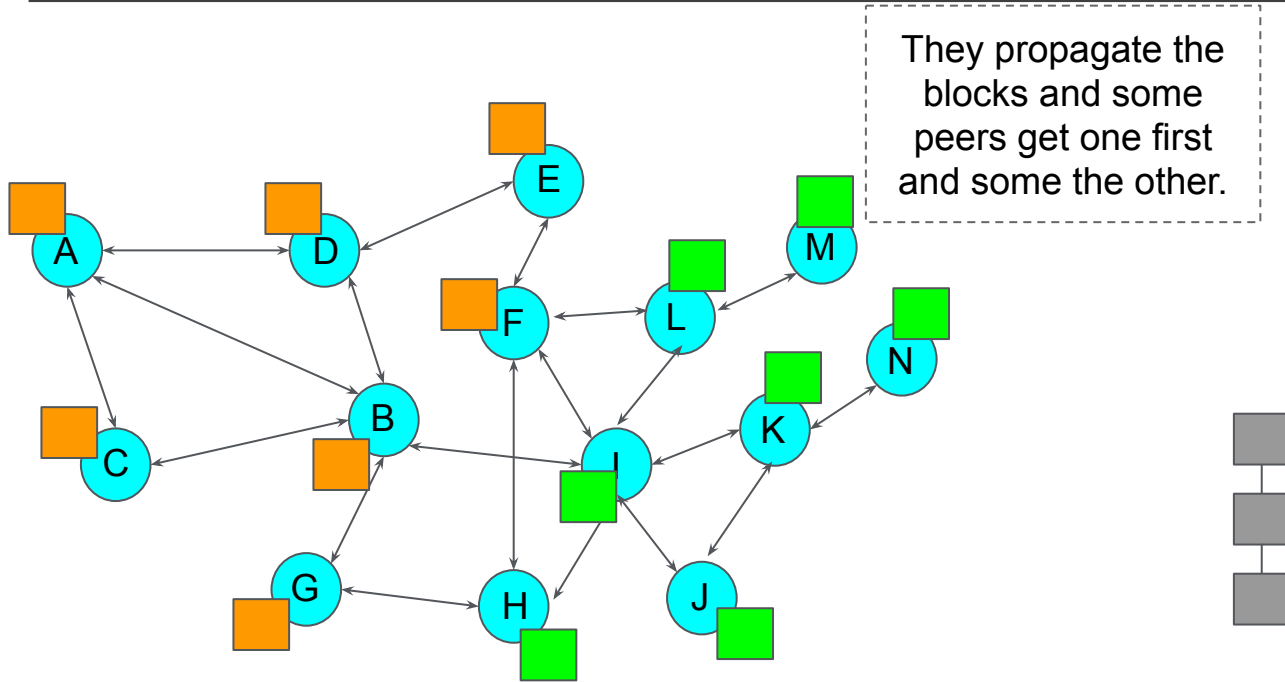


Nakamoto Consensus (2)

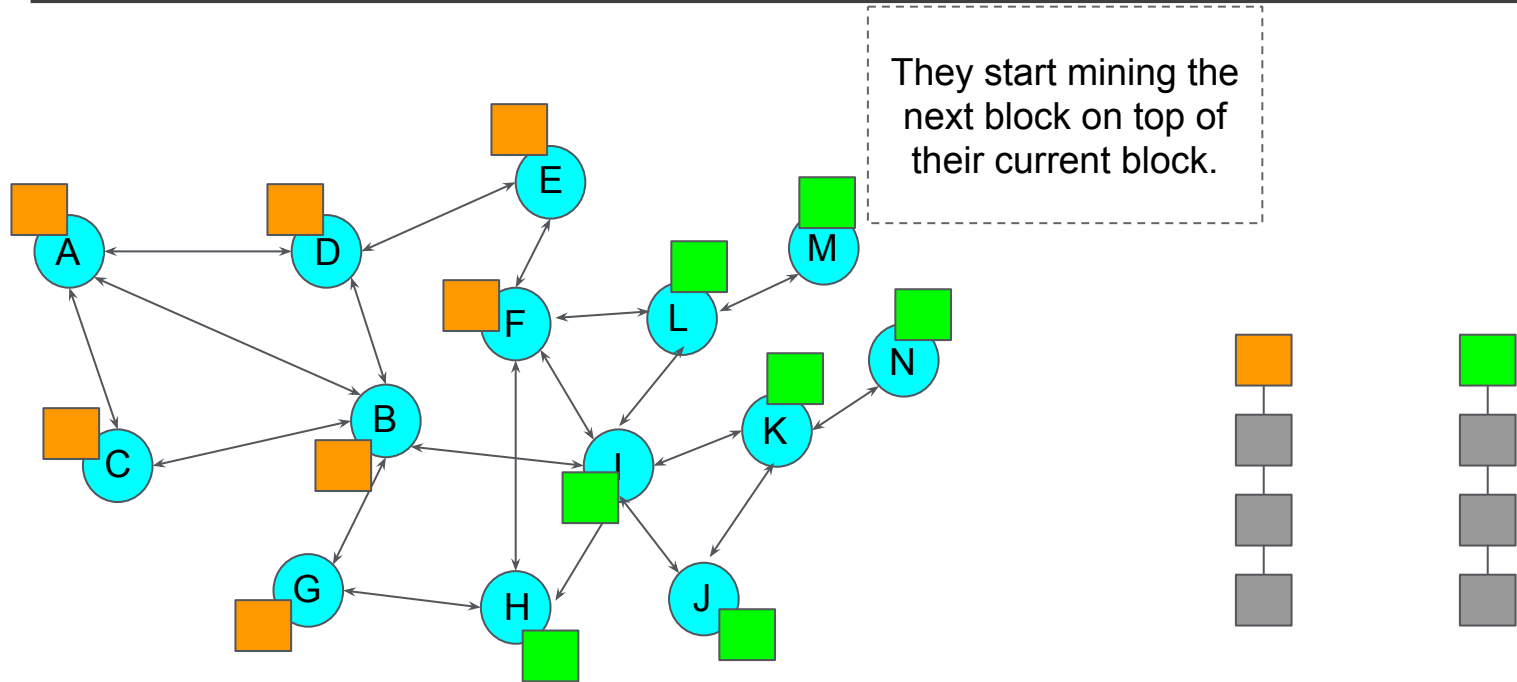
Two nodes find a solution at about the same time.



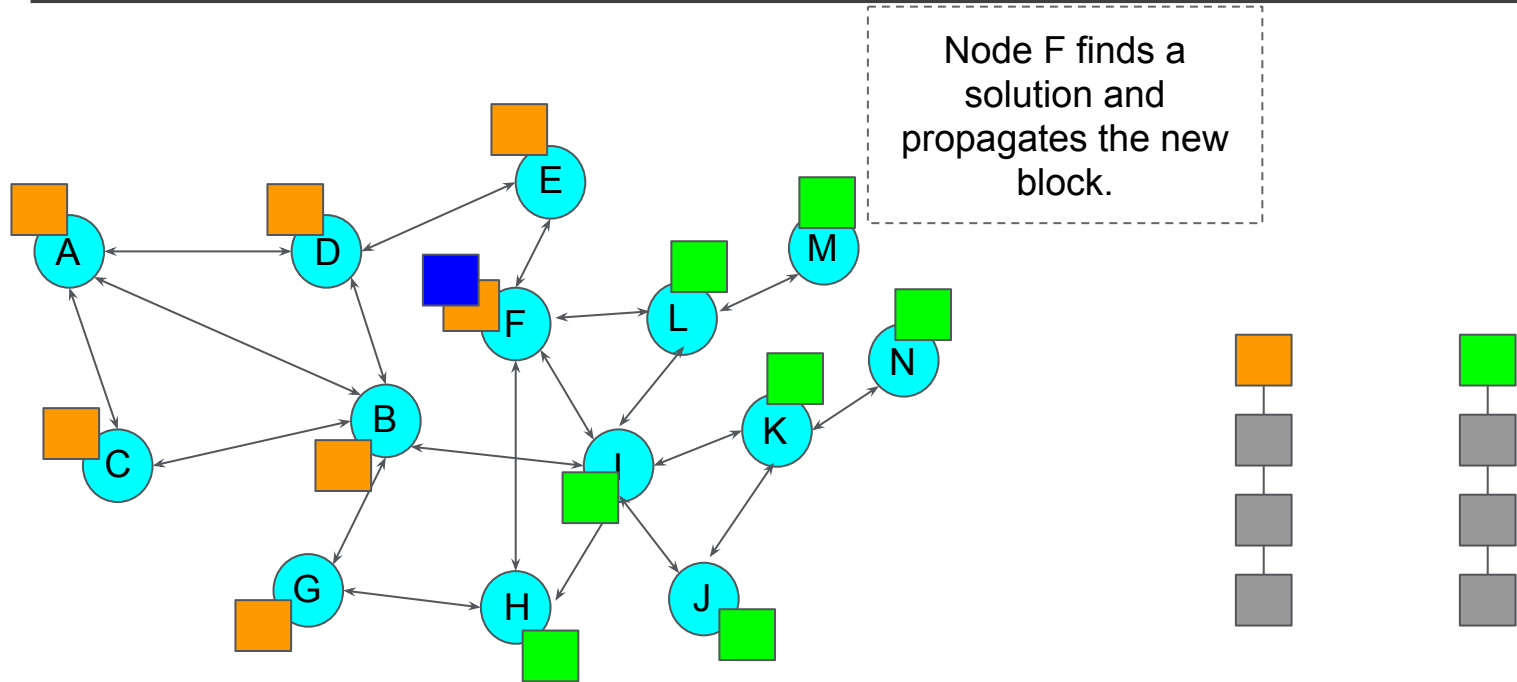
Nakamoto Consensus (2)



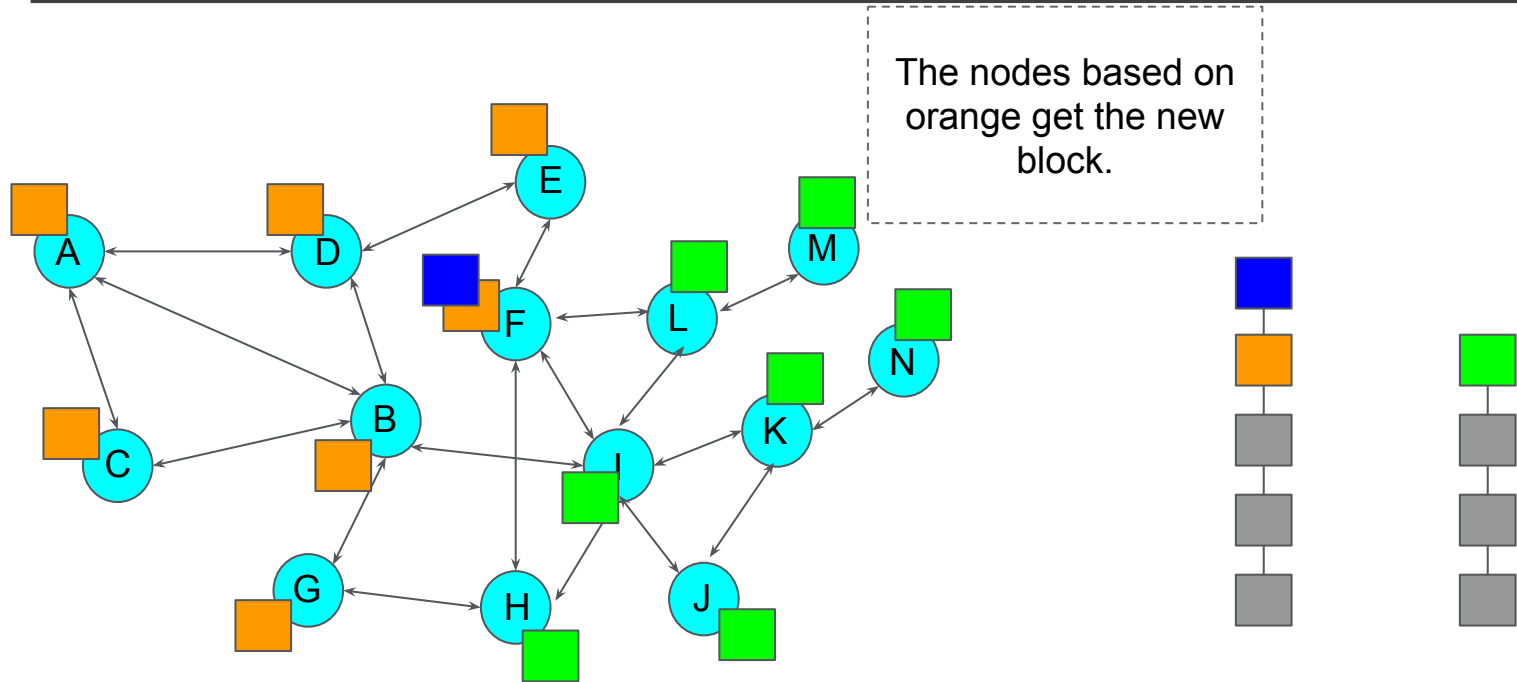
Nakamoto Consensus (2)



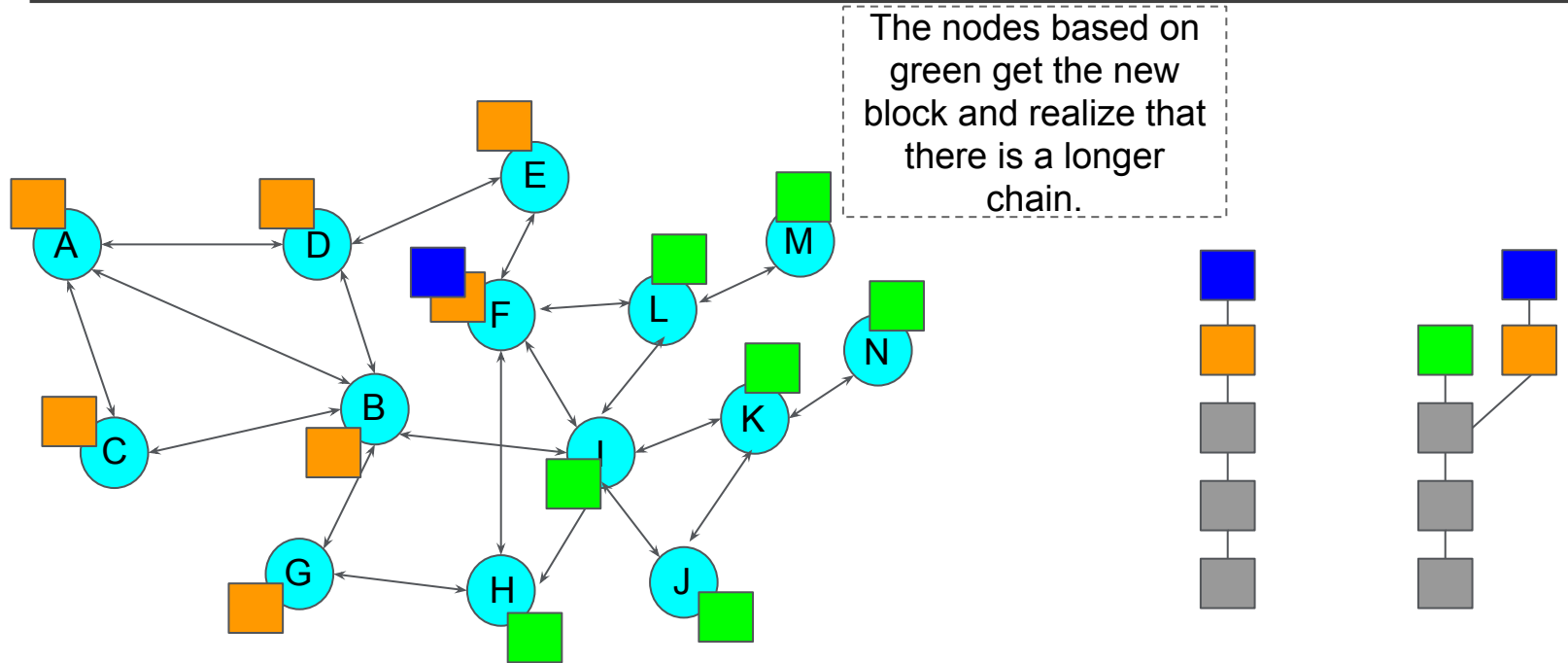
Nakamoto Consensus (2)



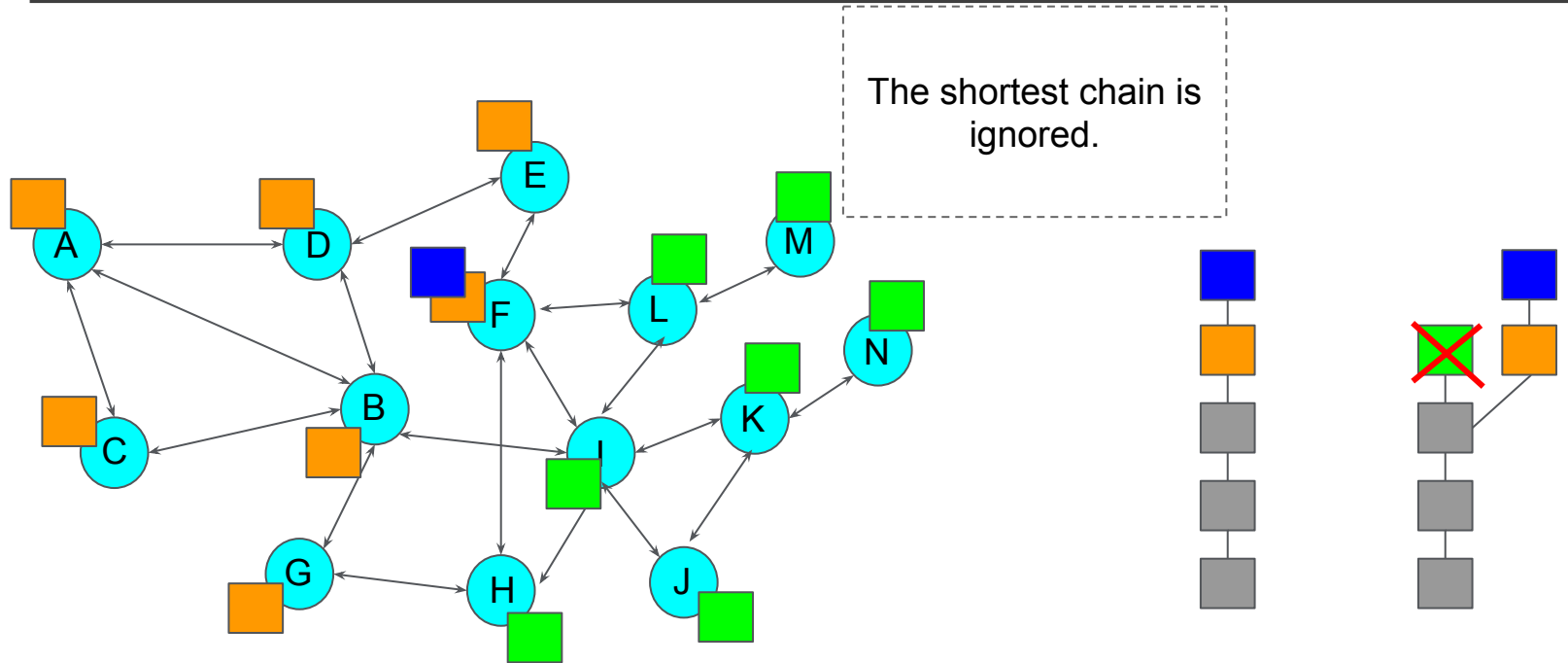
Nakamoto Consensus (2)



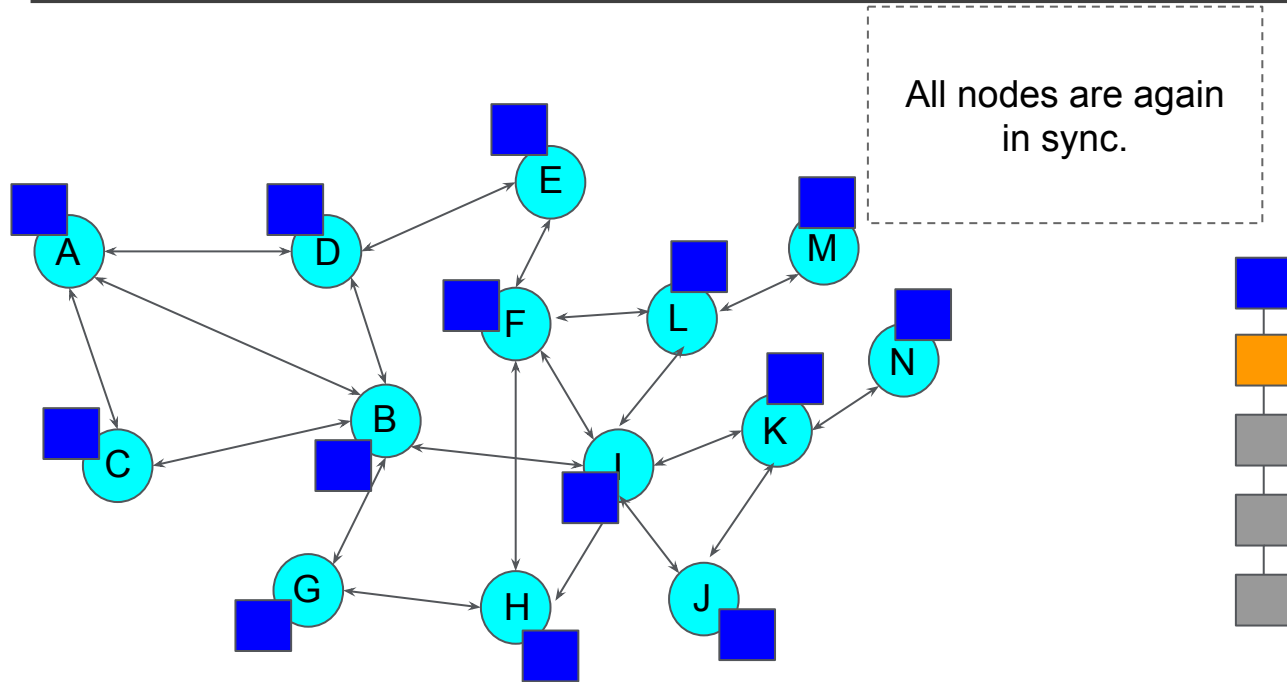
Nakamoto Consensus (2)



Nakamoto Consensus (2)



Nakamoto Consensus (2)



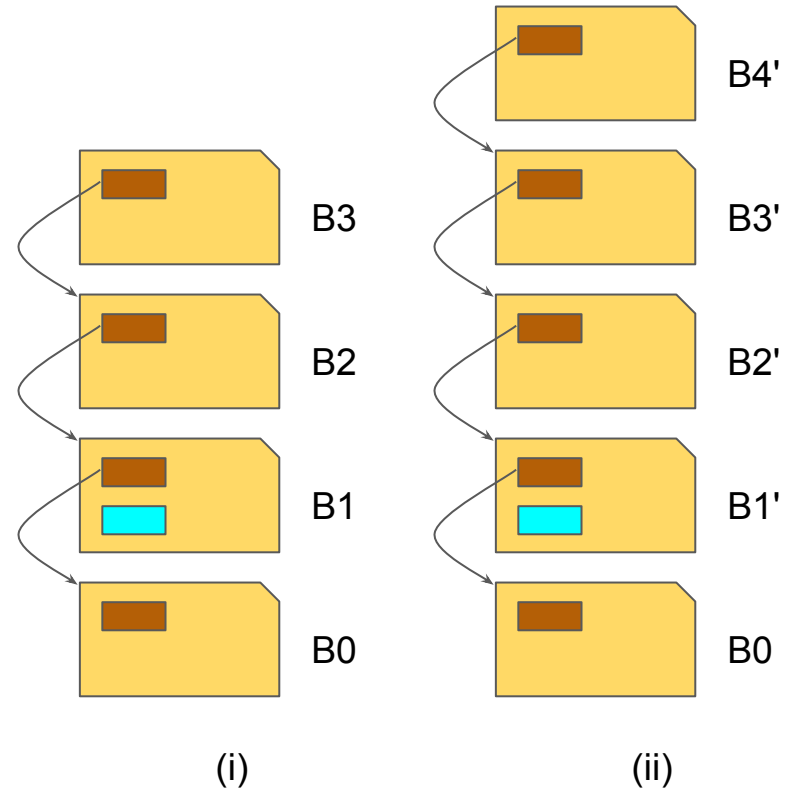
Blockchain and Trust (1)

Blocks are linked together by including the hash of the previous block* on the new block. E.g. the hash of B1 is included in the header of B2.

In our example a transaction in B1 (represented with the cyan box) has 3 confirmations. If an attacker wishes to attempt a double spend attack they will need to create a new B1' block with the modified transaction.

However, there are two more blocks on top of B1 and thus the attacker's block will be ignored since B1' will not be the longer chain. The attacker also needs to create B2', B3' and B4' to succeed in a double spend.

* More precisely it is the hash of the header, which represents the whole block.

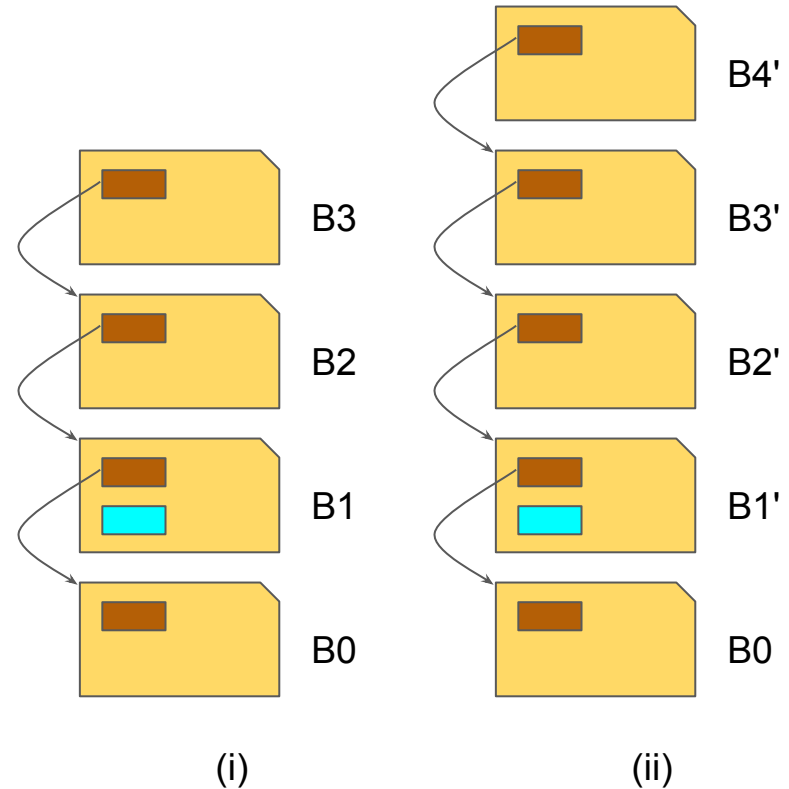


Blockchain and Trust (2)

To achieve that, the attacker will need to have the majority of the network's hash rate, which is what is typically called the 51% attack.

Achieving this kind of hash rate and sustaining it would require extravagant amounts of funds to accommodate for the mining hardware and operational costs and thus it would not be easily feasible.

This is even more evident when one considers what is possible with such an attack: potential censorship and double spends. Even with such an attack the funds on all the Bitcoin addresses are safe as is the historical records of the transactions; the former are secured by strong cryptography while the latter would require much more hash rate to modify them.

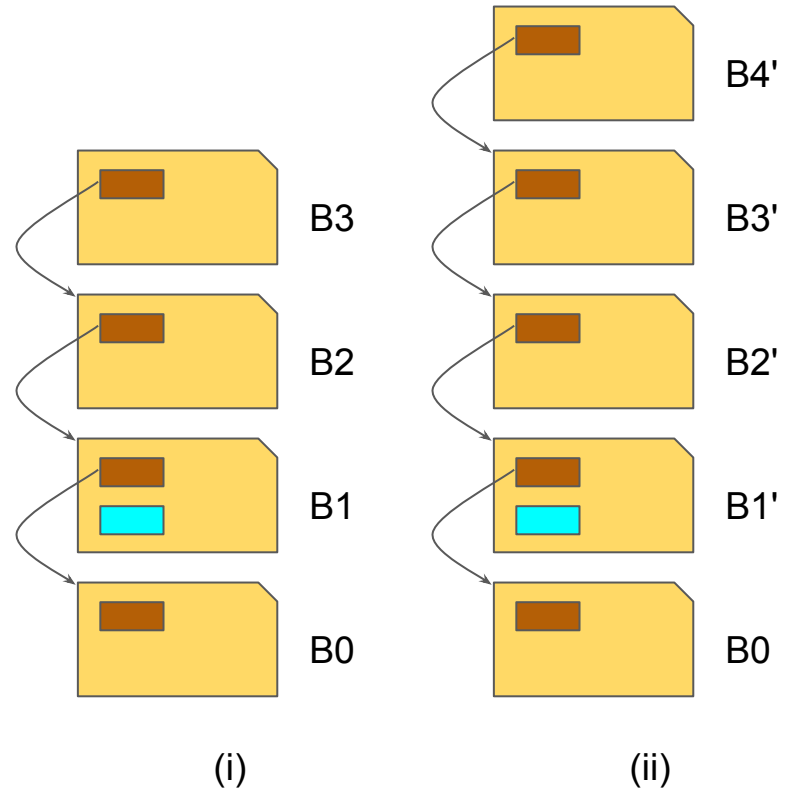


Blockchain and Trust (3)

Bitcoin security model is based on game theory principles and proper incentives. Economically speaking only a very irrational entity would make such an attack since setting up the environment for the attack would position the attacker in a very economically advantageous position, i.e. they will be earning a lot of money with the mined bitcoins.

Even though Bitcoin and Nakamoto Consensus provide us with some of the strongest probabilistic guarantees it is theoretically possible to be influenced by malevolent actors.

Until now the network had been extremely resilient to any kind of attack and has proven its robustness and stability demonstrating that is currently the most immutable structure constructed by humans.



Section 5: Basic interaction with a node



Bitcoin software

The Bitcoin software includes several executables, one providing the core functionality and the other utility tools:

bitcoind:

The daemon server provides full peer functionality; includes a wallet. It provides a JSON-RPC API to talk to the node (ports: mainnet: 8332, testnet: 18332, regtest: 18443).

bitcoin-cli:

Provides a command-line interface to *talk* to the daemon server

bitcoin-qt:

Provides a graphical user interface to the Bitcoin peer and wallet (subset of the API as part of GUI but also provides a console for all calls)

bitcoin-tx:

Allows to create, parse or modify transactions

bitcoin-wallet:

Wallet related utilities



Bitcoin software configuration and dev. Environments (1)

The configuration file is `bitcoin.conf` and its default location depends on the operating system used (e.g. in linux system it is located at `~/ .bitcoin/bitcoin.conf`). An [example config](#) gives several possible options but some important ones for [development and testing](#) your application follow:

daemon=1

Runs the Bitcoin node in the background.

server=1

Allows JSON-RPC commands but only from localhost.

prune=1000

Only keep more recent blocks that fit in 1000 MiB. Pruning is not compatible with `txindex` and `rescan`.



Bitcoin software configuration and dev. Environments (2)

testnet=1

The Bitcoin node uses the testnet network for development (i.e. fake funds). If the option is missing or if it is '0' then mainnet (the real network) is used.

regtest=1

This is a local test environment. The blockchain starts at height 0 (genesis block) and we can trivially mine new blocks with the `generatetoaddress` command. This allows developers to also control the block creation and get fake funds immediately. Regtest uses testnet's network parameters (e.g. address prefixes, etc).

signet=1

New test network for development that adds an additional signature requirement for block validation. Signet is similar in nature to testnet, but more reliable and centrally controlled. Anyone can run their own unique signet for their testing purposes.



Bitcoin software configuration and dev. Environments (3)

Other important options include:

addnode=12.23.34.56

Also connect to specific peer (multiple `addnode`'s can be used). If no network is specified it will only apply to `mainnet`.

connect=98.76.54.32

Only connect to specific node (multiple `connect`'s can be used). If no network is specified it will only apply to `mainnet`.

rpcallowip=12.34.56.78

Allows JSON-RPC connections from this IP (default is `localhost`).

[testnet]

Specifies a `testnet` section. All options after this heading will apply only to `testnet`. Other sections are `mainnet` and `regtest`. Some options, like `addnode` and `connect` need to be defined in a section otherwise they only apply on `mainnet`.



JSON-RPC API Calls (1)

```
$ ./bitcoin-cli help

$ ./bitcoin-cli getblockcount
1128802

$ ./bitcoin-cli getbalance
1.51815479

$ ./bitcoin-cli getnewaddress "" legacy
mvBGdiYC8jLumpJl42ghePYuY8kecQgeqS

$ ./bitcoin-cli encryptwallet MyPaSsWoRd
wallet encrypted; Bitcoin server stopping, restart
to run with encrypted wallet. The keypool has been
flushed, you need to make a new backup.

$ ./bitcoin-cli walletpassphrase MyPaSsWoRd 120

$ ./bitcoin-cli backupwallet wallet.backup

$ ./bitcoin-cli importwallet wallet.backup
```

Next week we are going to examine the different kind of addresses. Bitcoin v0.20+ uses bech32 (or native segwit) addresses by default. The example above explicitly creates a legacy address.

```
$ bitcoin-cli getnetworkinfo
{
  "version": 200000,
  "subversion": "/Satoshi:0.20.0/",
  "protocolversion": 70015,
  "localservices": "0000000000000409",
  "localservicesnames": [
    "NETWORK",
    "WITNESS",
    "NETWORK_LIMITED"
  ],
  ...
}

$ bitcoin-cli getblockchaininfo
{
  "chain": "test",
  "blocks": 1887283,
  "headers": 1887283,
  "Bestblockhash": "000000000074e...9d44e05b4",
  "difficulty": 1420477.254893854,
  "mediantime": 1604662239,
  "verificationprogress": 0.9999999194957088,
  "initialblockdownload": false,
  "chainwork": "00000000000...a2762e8",
  "size_on_disk": 28640545955,
  "pruned": false,
  ...
}
```



JSON-RPC API Calls (2)

```
$ bitcoin-cli getmininginfo
{
  "blocks": 1887283,
  "difficulty": 1420477.254893854,
  "networkhashps": 131251268159888.9,
  "pooledtx": 9,
  "chain": "test",
  "warnings": "Warning: unknown new rules activated (versionbit 28)"
}
```

```
$ bitcoin-cli getwalletinfo
{
  "walletname": "",
  "walletversion": 130000,
  "balance": 3.22457944,
  "unconfirmed_balance": 0.00000000,
  "immature_balance": 0.00000000,
  "txcount": 353,
  "keypoololdest": 1597237767,
  "keypoolsize": 999,
  "hdseedid": "df71b88eac9079d73b3e9b3aa088952e487b9ae5",
  "unlocked_until": 0,
  "paytxfee": 0.00000000,
  "private_keys_enabled": true,
  "avoid_reuse": false,
  "scanning": false
}
```



JSON-RPC API Calls (3)

```
$ ./bitcoin-cli sendtoaddress mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS 0.01
ff8322626c21c5bdfa1d27f75a55a1cb1d3b764bb34063f64b38f0803c370c08

$ ./bitcoin-cli listunspent 2
[
  {
    "txid": "30d98980c56a139438f0c969ca30d4be2c7f865d098b905362263c5daca2afa7",
    "vout": 0,
    "address": "mgs9DLttzvWFkZ46YLSNKSZbgSNiMNUsdJ",
    "amount": 1.01452015,
    "confirmations": 20183,
    ...
  }
  ...
]

$ bitcoin-cli listlabels
[
  "",
  "Test1",
  ...
]

$ ./bitcoin-cli getaddressesbylabel ""
[ "mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS", ... ]
```

JSON-RPC API Calls (4)

```
$ ./bitcoin-cli createwallet "testwallet"
```

```
$ ./bitcoin-cli help createwallet
```

Use **help** with any command to get details and examples of how to invoke them!



Blockchain Explorer: Transaction Example

<https://blockchain.info/tx/c4888e83f3901757308eef9e6c708b688c742f0333cb8c623feabaa40505176>

BLOCKCHAIN

WALLET

CHARTS

STATS

MARKETS

API

Search for block hash, transaction, address, etc

Transaction View information about a bitcoin transaction

c4888e83f3901757308eef9e6c708b688c742f0333cb8c623feabaa40505176

1Bvtn9b2x3CRfsJHQxnrQRQAI2jpaJmA (0.80720892 BTC - Output)
18N6be2X4FGjCJMzFnFa32YT4dpQaDFKuG (0.00479743 BTC - Output)



9D2XU4JsvUwKnuN1HVvdtGW8q37jABSJaF - (Unspent) 0.8 BTC
1HADUH7Hs5p9LxLBbYUENwjmU8gnKLdFY1 - (Unspent) 0.01000635 BTC

1 Confirmations

0.81000635 BTC

Summary

Size	372 (bytes)
Received Time	2017-07-06 08:52:52
Lock Time	Block: 474469
Included in Blocks	474484 (2017-07-06 08:53:29 + 1 minutes)
Confirmations	1 Confirmations
Relayed by IP	34.252.146.106 (whois)
Visualize	View Tree Chart

Inputs and Outputs

Total Input	0.81200635 BTC
Total Output	0.81000635 BTC
Fees	0.002 BTC
Fee per byte	537.634 sat/B
Estimated BTC Transacted	0.8 BTC
Scripts	Hide scripts & coinbase



Conclusions



Conclusions

- We went through the process of how a transaction is created in the Bitcoin network, the basic structure of a transaction and common types.
- We explained how transactions are propagated between nodes and how blocks are created from transactions.
- We have seen how a block is mined and how the mined block is propagated again through all the nodes to validate it and, if valid, add it to the blockchain.
- Finally, we saw some basic commands that we can give to a bitcoin node to create new addresses and send bitcoins to other addresses.



Self-assessment exercises and further reading



Self-assessment exercises

- Prepare a bitcoin environment by installing a Bitcoin node configured for testnet.
- Using bitcoin-cli create a new address
 - use a testnet faucet to get some test coins
- Using bitcoin-cli to send some test bitcoins to some of your classmates
 - share your testnet addresses via the forums
- Backup your wallet
- Go through the rest of the API and get familiar with more commands

You are welcome to use the forums to report issues, questions or your thoughts in general!



Further Reading

Bitcoin Programming Textbook (Ch.1), Kostas Karasavvas

<https://kkarasavvas.com/assets/bitcoin-textbook.pdf>

Mastering Bitcoin (Ch.2, Ch.10), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch02.asciidoc>

Bitcoin Developer Examples

<https://bitcoin.org/en/developer-examples>

(API calls with examples)

Bitcoin API calls

https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list

(A list of Bitcoin JSON-RPC API calls)

Running Bitcoin

https://en.bitcoin.it/wiki/Running_Bitcoin

(Running and configuring Bitcoin)





UNIVERSITY *of*
NICOSIA

The background is a dark blue, blurred image of a computer screen. It shows various lines of code, some in white and some in yellow, scattered across the frame. The text is out of focus, creating a bokeh effect. The overall aesthetic is technical and digital.

BLOC-521 Digital Currency Programming Cryptographic Keys Konstantinos Karasavvas



Objectives of Session

- Introduce cryptographic primitives used in Bitcoin
- Explain private/public keys and how to generate them

In this session we go through some basic cryptography needed to explain the keys used in Bitcoin and the rationale behind the process of their creation.



Agenda

- Hashing Algorithms
- Asymmetric Cryptography
- Bitcoin's Private keys
- Bitcoin's Public keys
- Conclusions
- Further reading



Section 1: Hashing Algorithms



Cryptographic Hash Function (1)

A **cryptographic hash** function is a hash function that takes an arbitrary block of data and returns a fixed-size bit string, the cryptographic *hash value*, such that any (accidental or intentional) change to the data will also change the hash value significantly.

Properties of a good cryptographic hash function:

- It is deterministic, i.e. the same block of data always returns the same hash
- It is quick to compute
- It is impossible to generate the block of data from the hash (one-way Fn)
- A small change to the data will change the hash so that it appears uncorrelated to the old hash value
- It is close to impossible to find two different blocks of data with the same hash value

Bitcoin is primarily using the SHA-256 function. The hash value is 256 bits or 32 bytes long. Each byte is usually represented by 2 hexadecimal numbers, and thus a SHA-256 function can be represented by 64 hexadecimal numbers.



Cryptographic Hash Function (2)

\$	sha256sum
Bitcoin	
deb10ca6fd85a5eba792ea8561da390635242f0c37c376f8eb7d7859adbffca9	-
\$	sha256sum
bitcoin	
61d520ccb74288c96bc1a2b20ea1c0d5a704776dd0164a396efec3ea7040349d	-

Cryptographic hash functions are very important in information security systems. They are used in digital signatures, message authentication codes and as ordinary (but more secure) hash functions to index data in hash tables, to uniquely identify files (bittorrent, IPFS), as checksums to detect accidental (or not) corruption of data, etc.

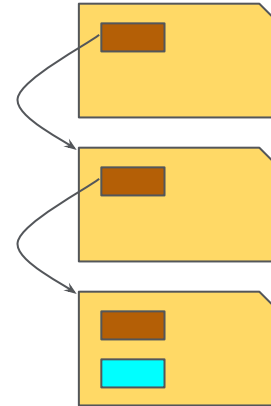
The result of a hash function is referred to as *digital fingerprints* or *hash values* or just *hashes*.



Cryptographic Hash Function (3)

In Bitcoin the SHA-256 hash function is used for:

- Block chaining & Integrity verification
 - When a new block is created its header contains the hash of the previous block, thus *chaining* the two blocks together.
 - Since peers follow the longest chain a Tx with more confirmations (chained blocks on top of the block that contains it) is much more difficult to alter/remove since to change the Tx successfully an attacker has to modify that block as well as all the blocks on top of it (or else the tampering will be detected immediately)
 - This is computationally infeasible to sustain (i.e. a 51% attack)



Cryptographic Hash Function (4)

In Bitcoin the SHA-256 hash function is used for:

- Proof-of-Work (hashcash) cost-function (mining)
 - The Proof-of-Work puzzle involves finding a hash of the new block that is less than a certain value
- Generation of Bitcoin addresses
 - Used to improve security and privacy
 - For creating addresses another hashing algorithm, [RIPEMD-160](#), is also used



Section 2: Asymmetric Cryptography



Asymmetric Cryptography

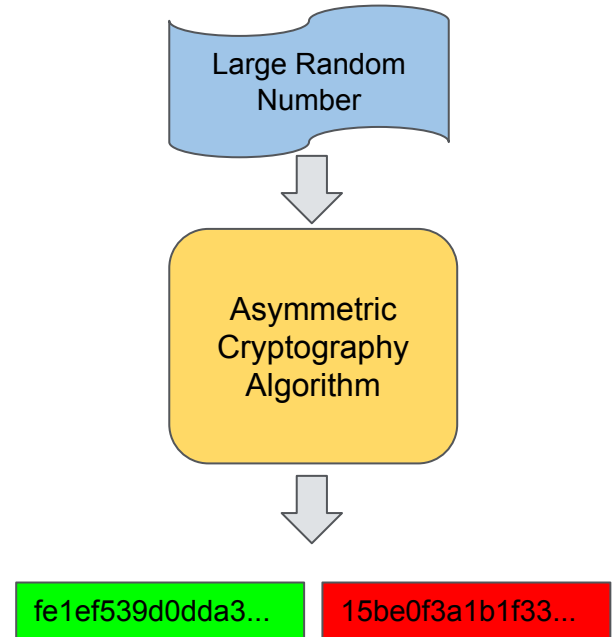
Asymmetric cryptography or public key cryptography is a cryptographic system that uses pairs of keys with a specific mathematical relation. In each pair there is a private key that should remain private and a public key that can be freely shared. Between two participants this allows:

Encryption

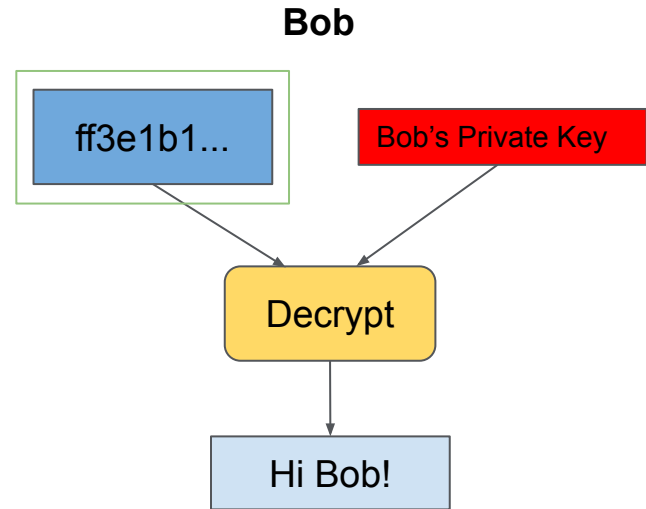
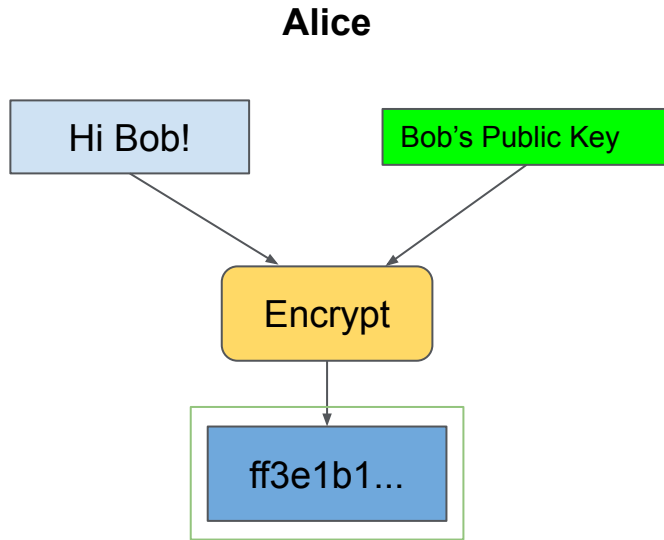
Alice can *encrypt* a message with Bob's public key and send it to Bob. Only the owner of the corresponding private key can decrypt and view the message.

Authentication / Digital Signatures

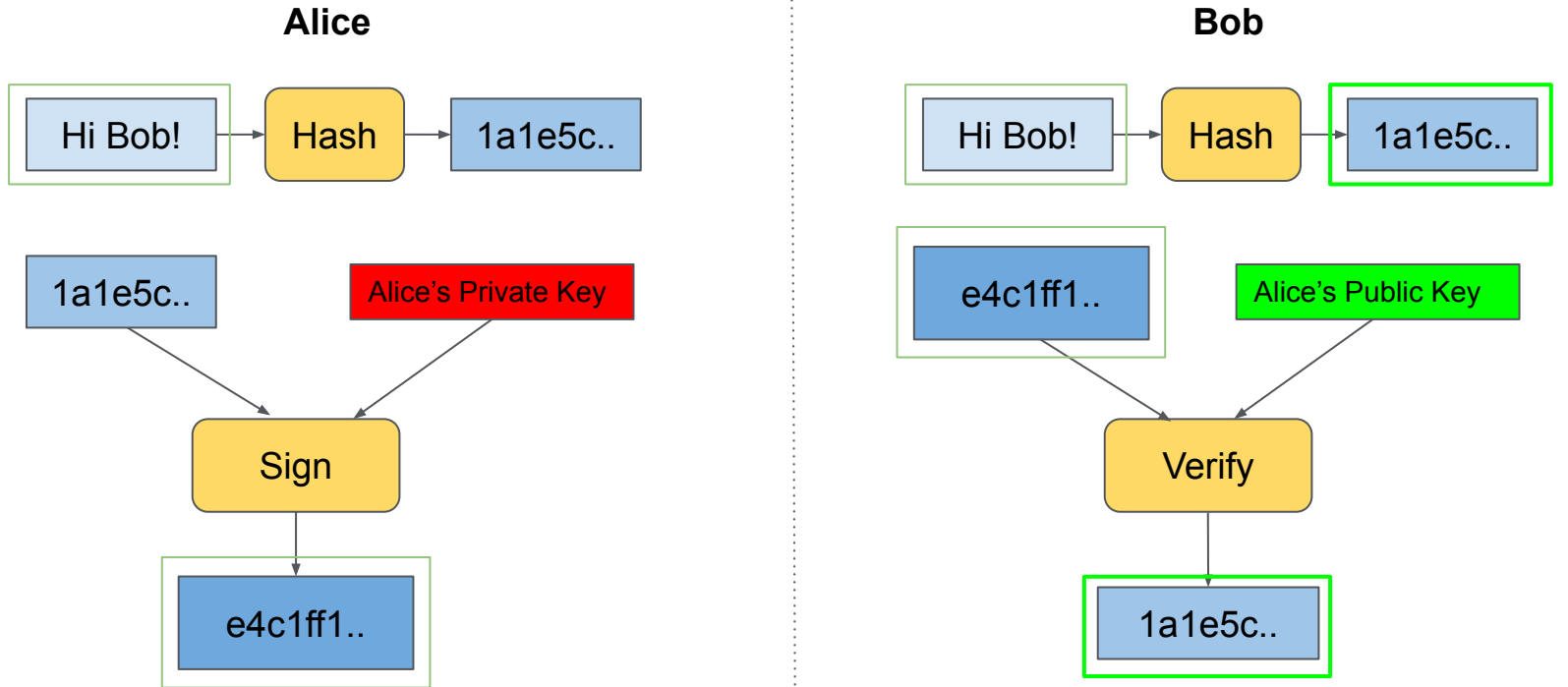
Alice can *sign* a message using her private key and send it to Bob. Anyone can view the contents and verify the signature using Alice's public key, thus ensuring that it was indeed Alice that send the message.



Encryption / Decryption



Digital Signatures / Verification



Public Key Cryptography in Bitcoin

Encryption is not used at all.

Digital signatures are used to sign transactions in order to authenticate that you are the owner of the coins you wish to transfer.

Bitcoin uses the [Elliptic Curve Digital Signature Algorithm](#) (ECDSA) to create its private-public key pairs. The exact elliptic curve parameters used in Bitcoin are defined by [secp256k1](#).

In ECDSA a private key can be used to calculate the corresponding public key, and since the address is calculated from the public key, if you hold a private key securely you effectively have everything.



Section 3: Bitcoin Private Keys



Private Keys

The ECDSA private key in Bitcoin is just a random number consisting of 256 bits or 32 bytes or 64 hexadecimal digits. Nearly all 256-bit numbers can be valid private keys as specified in secp256k1.

The most common format used to display a private key is *Wallet Import Format* (WIF) or a WIF-compressed (WIFC); both are a [Base58Check encoding](#) of the ECDSA key; [Base58](#) with version prefix to specify the network and a 32-bit checksum.

A WIF-compressed adds an extra byte (0x01) at the end of the ECDSA key before the Base58Check encoding. It specifies whether the public key (and by extension addresses) will be compressed or not. By default most wallets use WIFC format in order to reduce the size of the blockchain.

```
$ ./bitcoin-cli dumpprivkey mg6KkpbddyFkwzCFzmnza3yZAUj2yPhoKd  
cNg68oFh99vkg5FRkegvx11jq5w9bxzBaDan9ZLUfZeMr4Vn6dii
```



Private Keys: formats (1)

	Mainnet		Testnet	
ECDSA HEX	64 digits number		64 digits number	
ECDSA HEX-C	Above number + "01"		Above number + "01"	
	Version Prefix	Base58 Prefix	Version Prefix	Base58 Prefix
WIF	128 0x80	5	239 0xef	9
WIF-Compressed	128 0x80	K or L	239 0xef	c

Check it out by entering a testnet private key on "Wallet Details" at: <https://www.bitaddress.org> or <https://www.bitaddress.org?testnet=true>



Private Keys: formats (2)

ECDSA in HEX	0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b
ECDSA in HEX-C	0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b 01
WIF	91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p
WIFC	cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo

Use [libbitcoin-explorer](#) to experiment with private keys' formats. Decimal [prefixes](#) 128 (0x80) and 239 (0xef) are used for private keys of mainnet and testnet respectively.

```
$ ./bx base58check-encode --version 239 0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b  
91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p
```

```
$ ./bx base58check-encode --version 239  
0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b01  
cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo
```

Calculate WIF/WIFC

To calculate to WIF/WIFC we need to apply the following algorithm:

```
key_bytes = (32 bytes number) [ + 0x01 if compressed ]
network_prefix = (1 byte version number)
data = network_prefix + key_bytes
data_hash = SHA-256( SHA-256( data ) )
checksum = (first 4 bytes of data_hash)
wif = Base58CheckEncode( data + checksum )
```

All byte sequences should be treated as big-endian.
To validate a WIF/WIFC key one can [Base58CheckDecode](#), remove the checksum and double SHA-256 the remainder, which should be equal to the checksum.



Section 4: Bitcoin Public Keys



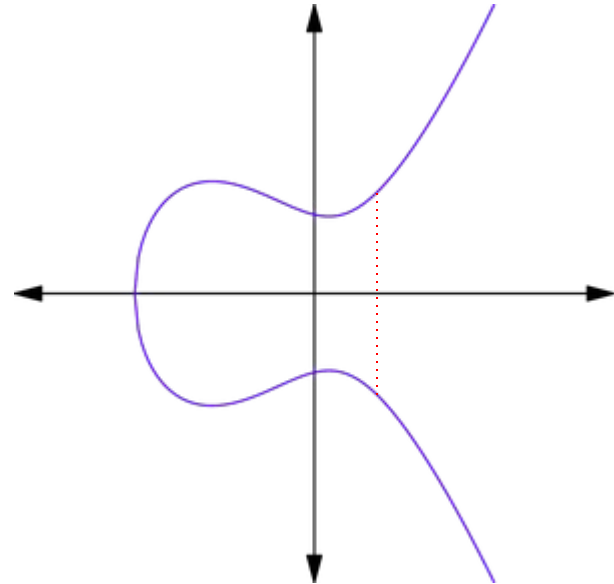
Public Keys

A public key is generated from the private key using *elliptic curve multiplication*, a one-way function, that generates the public key.

Effectively the public key is a point P in the elliptic curve ($y^2 = x^3 + 7$), $P = (x,y)$. Both x and y are 32 bytes, thus the key is 64 bytes.

An encoded uncompressed public key is 65 bytes long since it has the two points (32 bytes each) concatenated and a prefix of $0x04$ to specify an uncompressed public key.

An encoded compressed public key is 33 bytes long and has only the x coordinate with a prefix of $0x02$ (y) or $0x03$ ($-y$). It turns out that the y coordinate can only take 2 values (positive/negative) for a specific x .



Public Keys: example

WIF	91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNI4nVfiVwkMPfA9p
Uncompressed Public Key	04c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7ad dc4cbba6656a4be4bc6933a6af712b897a543a09c4b899e5f7b943d38108a8
WIFC	cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo
Compressed Public Key	02c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7

```
$. /bx wif-to-public 91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNI4nVfiVwkMPfA9p  
04c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7ad  
dc4cbba6656a4be4bc6933a6af712b897a543a09c4b899e5f7b943d38108a8
```

```
$. /bx wif-to-public cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo  
02c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7
```



Conclusions



Conclusions

- We introduced basic cryptography required to understand Bitcoin
- We explained what are private keys and public keys in Bitcoin, how they are generated



Further Reading



Further Reading

Cryptographic Hashes

https://en.wikipedia.org/wiki/Cryptographic_hash_function

Asymmetric Cryptography

https://en.wikipedia.org/wiki/Public-key_cryptography

Understanding Elliptic Curves

<https://eng.paxos.com/blockchain-101-elliptic-curve-cryptography>

Mastering Bitcoin (Ch.4), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc>

(Bitcoin keys and addresses)





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming Addresses and Wallets Konstantinos Karasavvas



Objectives of Session

- Explain addresses and how to generate them
- Describe the different types of Bitcoin wallets
- Setup programming environment and create first programs

In this session we introduce Bitcoin's addresses and describe the rationale behind the process of their creation. Finally go through the different wallet types.



Agenda

- Bitcoin's Addresses
 - Vanity Addresses
- Wallets
- Conclusions
- Self-assessment exercises and further reading



Section 1: Bitcoin Addresses



Addresses

Addresses can be shared to anyone who wants to send you money. They are typically generated from the public key, consist of a sequence of characters and digits and start with 1 for the mainnet and with m or n for testnet.

An address typically represents the owner of a private/public pair but it can also represent a more complex script as we will see in future sessions.

Notice that we do not share the public key as one would expect in public key cryptography but rather the address, which derives from the public key. Some benefits are:

- shorter addresses
- quantum computer resistant
 - until you spend from an address your public key will never be visible from an outsider and since the address is hashed from the public key not even quantum computers could brute force to get the public key and then the private key...



Calculate Address

To calculate an address we need to apply the following algorithm:

```
version = (1 byte version number)
keyHash = RIPEMD-160( SHA-256( publicKey ) )
data = version + keyHash
dataHash = SHA-256( SHA-256( data) )
checksum = (first 4 bytes of dataHash)
address = Base58CheckEncode( data + checksum )
```

All byte sequences should be treated as big-endian. To validate a bitcoin address one can Base58CheckDecode, remove the checksum and double SHA-256 the remainder, which should be equal to the checksum.



Another Python library example

```
$ pip install bitcoin-utils
```

```
from bitcoinutils.setup import setup
from bitcoinutils.keys import PublicKey

setup('testnet')
pub = PublicKey.from_hex('04c1acdac799fb0308b4b6475ddf7967676759d31484ab555554'\
                        '82472f3bc7c3e7addc4cbba6656a4be4bc6933a6af712b897a543a09c4'\
                        'b899e5f7b943d38108a8')
pubc = PublicKey.from_hex('02c1acdac799fb0308b4b6475ddf7967676759d31484ab55555'\
                          '482472f3bc7c3e7')

print(pub.get_address(compressed=False).to_string())
print(pubc.get_address().to_string())
```

```
$ python addr.py
n2JjAgC6UqFf8DvsZXhWcyNzm8w8YKj7MQ
n42m3hGC52QTChUbXq3QAPVU6nWkG9xuWj
```



Addresses and prefixes

Uncompressed Public Key	04c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7ad dc4cbba6656a4be4bc6933a6af712b897a543a09c4b899e5f7b943d38108a8
Uncompressed Address	n2JjAgC6UqFf8DvsZXhWcyNzm8w8YKj7MQ
Compressed Public Key	02c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7
Compressed Address	n42m3hGC52QTChUbXq3QAPVU6nWkG9xuWj

	Mainnet		Testnet	
	Version prefix	Base58 prefix	Version prefix	Base58 prefix
P2PKH Address	0 0x00	1	111 0x6f	m or n
P2SH Address	5 0x05	3	196 0xc4	2



Bech32 Encoding addresses

Segregated Witness (segwit) is a consensus change that was activated in August 2017 and introduces an update on how transactions are constructed. It introduces two new transaction types, Pay-to-Witness-Public-Key-Hash (**P2WPKH**) and Pay-to-Witness-Script-Hash (**P2WSH**). These new transaction types are going to be explained in detail in the following sessions.

With regard to addresses, these new transaction types use [Bech32 encoding](#) instead of Base58Check. This new encoding is defined in [BIP-173](#) where an explanation is provided of why it was introduced and what its benefits are.

Bech32 addresses start with “bc” on mainnet and “tb” on testnet, e.g:

```
bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4  
tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsj
```

or

```
bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3  
tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7
```



Vanity Addresses (1)

These are normal addresses that contain a specific string. They are calculated randomly by creating random private keys and then check if the corresponding address starts with that string, e.g. 1KK.

```
import random
from bitcoinutils.setup import setup
from bitcoinutils.keys import PrivateKey

setup('mainnet')
vanity_string = '1KK'
found = False
attempts = 0

while(not found):
    p = PrivateKey(secret_exponent = random.getrandbits(256))
    a = p.get_public_key().get_address()
    print('.', end='', flush=True)
    attempts += 1
    if(a.to_string().startswith(vanity_string)):
        found = True

print("\nAttempts: {}".format(attempts))
print("Address: {}".format(a.to_string()))
print("Secret Key: {}".format(p.to_wif()))
```

Can you spot any potential issues?

Vanity Addresses (2)

You will notice that it takes some time even for a short string. Legacy addresses always start with `1` so we can disregard that. Since addresses use base58 each character will take an average of 58 attempts to be found. The next character an addition 58 attempts (thus 58×58). We can generalize with 58^n where n is the number of characters the vanity address should start with.

There are efficient implementations for calculating vanity addresses in C, Go, Rust or other compiled system languages that will calculate much faster than our simple example above but still to create `1Kostas` it will require 38,068,692,544 attempts (58^6). That will take considerable time regardless of the efficiency of the program or the hardware used.

In practice, these large vanity addresses are created via vanity address *pools*. Such pools have specialized hardware (i.e. mining hardware) that can create vanity addresses fast, albeit for a fee. However, how can they send you your private key that corresponds to the vanity address without them knowing it?

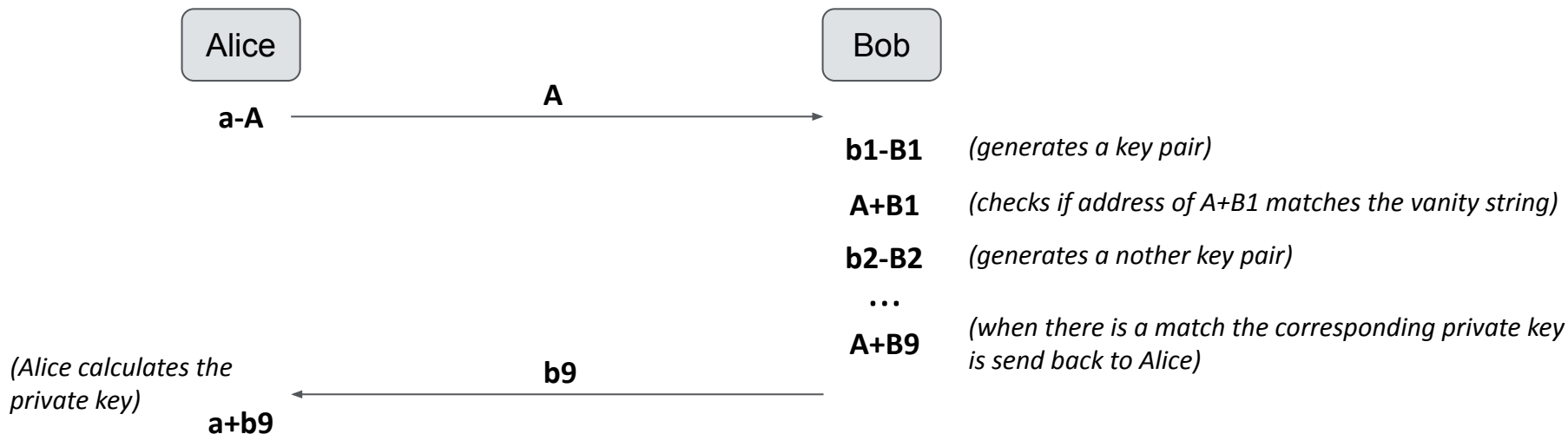


Vanity Address Pools

Vanity address pools take advantage of an elliptic curve cryptography property in which the public key of the sum of two public keys corresponds to the private key of the sum of the corresponding public keys. For example consider Alice having the key pair $a-A$ and Bob the key pair $b-B$, then:

$A+B$ will produce the public key of the $a+b$ private key.

Consider that Alice wants to use a pool operated by Bob to get a vanity address.



Section 2: Wallets



Wallets

A wallet is software that allows us to manage the private and public keys as well as our Bitcoin addresses. They usually have functionality to send bitcoins, check balances, create contact lists and other. Usually a key (i.e. address) is used only once.

Depending on how the private keys are handled there are two types of wallets:

Nondeterministic

All the private keys on the wallet are just randomly generated. Several private keys are pre-generated and new keys are created if needed. If you backup your wallet and then create new keys, you will need to backup your wallet again.

Deterministic

A seed is used to create a master private key, which can be used to create all other private keys (thus public keys and addresses as well). If you backup your seed you are safe no matter how many keys you use since all can be generated from the seed.



HD Wallets

Deterministic wallets are defined in *Bitcoin Improvement Proposals** (BIPs) [32](#), [43](#) and [44](#).

Creating the master keys requires a *root seed* comprising of either 128, 256 or 512 bits as generated by a (*Pseudo*) *Random Number Generator* or (P)RNG. The seed is passed as input to HMAC-SHA512 algorithm to create a 512 bit hash which is split in half. The first 256 bits (I_L) is the *master private key* and the last (I_R) 256 bits is the *master chain code*.

$$I = \text{HMAC-SHA512}(\text{Key} = \text{"Bitcoin Seed"}, \text{Data} = \text{root_seed})$$

$$m_{\text{privkey}} = I_L$$

$$m_{\text{chaincode}} = I_R$$

As expected, m_{pubkey} can be generated by m_{privkey} .

* BIPs are design documents that provide a way of communicating and discussing new ideas. They are used to introduce/specify new features for Bitcoin.

HD Wallets: Child Private Key Derivation (CKD)

As expected the master private key can be used to generate the master public key. Then the master key can be used to derive 2^{32} number of child keys by appending a 4 bytes index to the Data parameter of HMAC-SHA512. The chain code is used as the Key (entropy/salt).

The process to create child private keys is the following:

$$I = \text{HMAC-SHA512}(\text{Key} = m_{\text{chaincode}}, \text{Data} = m_{\text{pubkey}} \parallel \text{index})$$
$$k = I_L + m_{\text{privkey}}$$
$$c = I_R$$

|| is string concatenation
+ is point addition according to elliptic curve operations

The number of children depends on the size of index; 2^{32} private keys. For each child private key we can repeat the process to produce an immense amount of keys. Derived keys are indistinguishable from random keys. However, if you need to use the key to further derive more keys you also need the chain code. The combination of key with chain code is called *extended private key*.

Extended private keys start with xprv (mainnet) or tprv (testnet) and are 78 bytes.



HD Wallets: Child Public Key Derivation (CKD)

Interestingly, it is also possible to create child public keys only with the extended public key.

The process to create child public keys does not involve the parent private key and is the following:

$$I = \text{HMAC-SHA512}(\text{Key} = m_{\text{chaincode}}, \text{Data} = m_{\text{pubkey}} \parallel \text{index})$$
$$k = I_L + m_{\text{pubkey}}$$
$$c = I_R$$

The number of children depends on the size of index; 2^{31} public keys (only non-hardened keys can be used to derive child public keys - see next slide). For each child public key we can repeat the process to produce an immense amount of keys. Derived keys are indistinguishable from random keys. However, if you need to use the key to further derive more public keys you also need the chain code. The combination of key with chain code is called *extended public key*.

Extended public keys start with xpub (mainnet) or tpub (testnet) and are 78 bytes.



HD Wallets: Hardened Child Private Key Derivation

Accessing an xpub only allows the creation of child public keys but the xpub contains the chain code. Unfortunately there is a security risk in case a child private key is compromised. The latter together with the parent chain code can be used to generate all the child private keys. To circumvent this issue we can use a *hardened* CKD function which “breaks” the relationship between parent public key and child chain code; thus extended public keys cannot be used to derive child public keys.

$$I = \text{HMAC-SHA512}(\text{Key} = m_{\text{chaincode}}, \text{Data} = 0x00 \parallel m_{\text{privkey}} \parallel \text{index})$$
$$k = I_L + m_{\text{privkey}}$$
$$c = I_R$$

where $2^{32} > \text{index} \geq 2^{31}$.

The resulting chain code is different than the normal and cannot be used to compromise other private keys. The derived hardened keys can be used with normal derivation to create normal keys once again. The hardened key provides a barrier even if the normal keys below have been compromised.



HD Wallets: Derivation Paths and BIP 43

To specify the derivation path we typically use the following notation:

m: master private key

M: master public key

/i: non-hardened derivation at index i (from 0 to $2^{31}-1$)

/i' hardened derivation at index i' (from 2^{31} to $2^{32}-1$)

Examples:

M/0 : the first child public key

m/0'/9 : the 10th child of the first hardened child

m/0'/11/2/5/1 : ...

BIP 43 defines the structure of m/purpose' to act as namespaces for later uses. The first index (0) is already taken as the default account from BIP 32.



HD Wallets: Derivation Paths and BIP 44

BIP 44 defines the structure:

m / purpose' / coin_type' / account' / change / address_index

Where:

purpose' is 44' (complying to BIP 43)

coin_type' specifies the type of cryptocurrency coin (Bitcoin: 0', Testnet Bitcoin: 1', Litecoin: 2')

account' allows for sub-accounts per coin type

change can be either 0 for external chains (visible like payment addresses) and 1 for internal chains (not visible like change addresses)

address_index is the actual address index

Examples:

M/44'/1'/0'/0/0 : the first receiving public key for the first Bitcoin testnet account

m/44'/0'/2'/1/31 : the 32nd private key of the third Bitcoin change account

m/44'/2'/1'/0/2 : ...

HD Wallets: Use Cases

As a reminder, the main difference between non-hardened and hardened keys is that the former allows for extended public keys to generate child public keys but the latter does not.

The following use cases are irrespective of BIPs 43 and 44.

Wallet sharing:

When two systems both need to access a single shared wallet (same pool of addresses)

Audits:

Providing an extended public key will allow to audit all public addresses (of that sub-tree)

Per-office balances:

Each office will have its own extended private key and the company *owners* the master key

Unsecure web server:

E-commerce server is only given an extended public key so it can create receiving addresses but even if it is compromised there are no private keys.

Business-2-Business:

When business partners often transfer money between them one can provide an extended public key to the other so that the latter can create addresses on his own.



ypub and zpub

xpriv and **xpub** are used to denote that the final addresses are legacy P2PKH and will be prefixed with 1

ypriv and **ypub** are used to denote that the final addresses are nested segwit P2SH(P2WPKH) and will be prefixed with 3

zpriv and **zpub** are used to denote that the final addresses are native segwit P2WPKH and will be prefixed with bc1



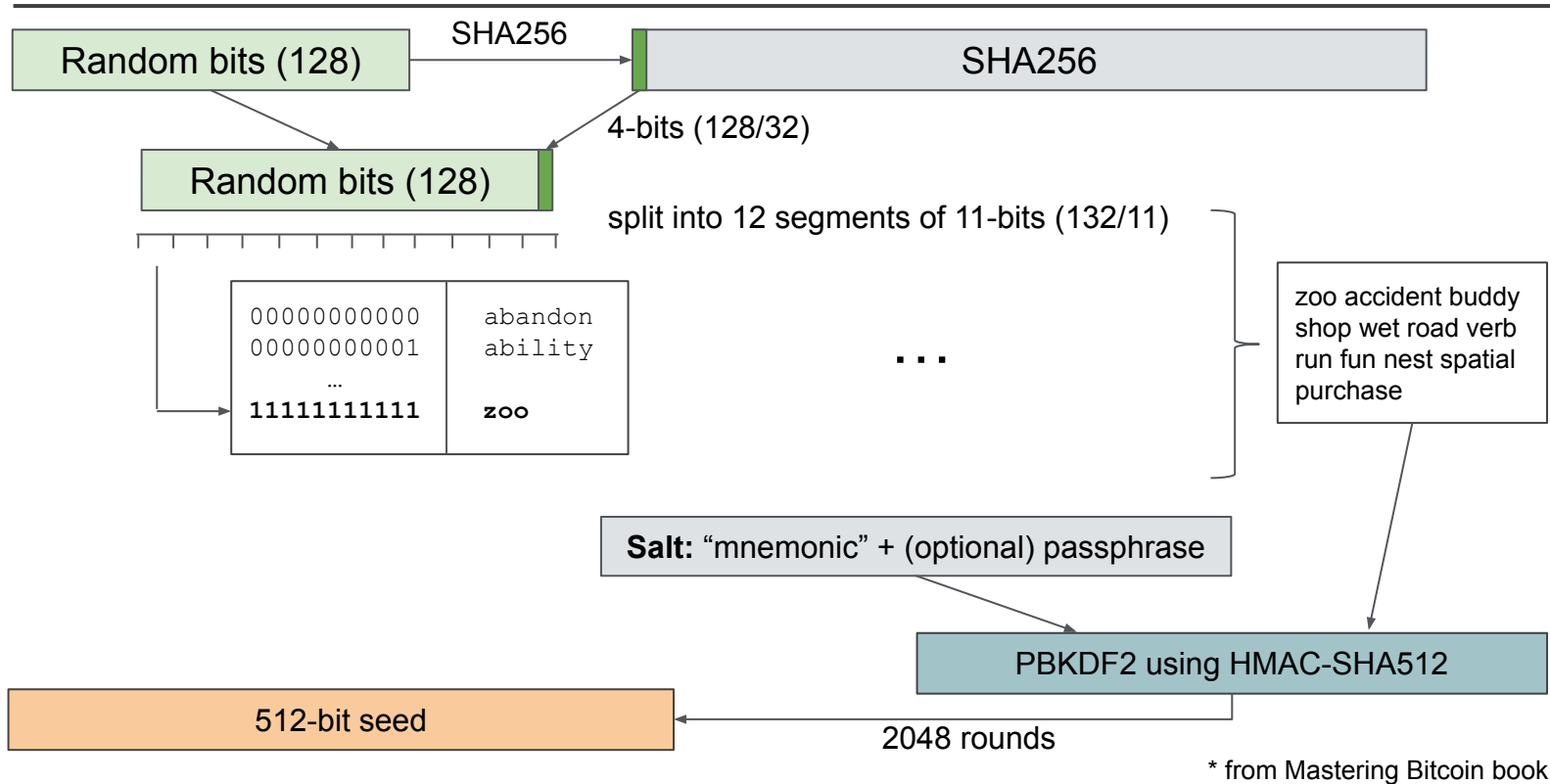
Mnemonic codes

Mnemonic codes are just a sequence of English words that are typically used to create the root seed of deterministic wallets, e.g. by hashing them. It is easier to memorize or write down those words than a sequence of random hexadecimal numbers.

Mnemonic codes are defined in [BIP-39](#) which includes several [wordlists](#) (2048 words per language).

- a PRNG will produce a random sequence of data (128-256 bits)
- the sequence will be hashed and the first bits will be added as a checksum
- the resulting sequence will be split every 11-bits
- each 11-bits correspond to a specific word from the wordlist
- the result will be the mnemonic which will be 12-24 words
- the words plus some *salt* will then be passed to a key-stretching function (PBKDF2 using HMAC-SHA512), to produce the final 512-bit seed.

Mnemonic codes (cont.)



* from Mastering Bitcoin book

Hardware Wallets

Hardware wallets is a secure and easy way to store to store bitcoins. The hardware wallet creates a mnemonic code itself. It uses associated software in a computer or a mobile phone to interact with a user for creating transactions.

Interaction is achieved using the Bitcoin [Hardware Wallet Interface](#) (HWI), a Python library and command-line tool that provides an ad-hoc standard for achieving this in a consistent way. Most hardware wallets are using this.

Using the software a user requests a transaction from the hardware wallet. The wallet receives the request and asks the user to confirm using a simple UI in the device. If confirmed, the hardware wallet signs the transaction with the corresponding private key.

Private keys never leave the wallet and are never displayed.



Paper Wallets and Encryption (1)

Paper wallets is a way to store bitcoins offline in a physical document. It usually contains only the private key and the Bitcoin address together with their respective QR codes.

Paper wallets can be created and printed using sites like: www.bitaddress.org and bitcoinpaperwallet.com.

The private key is in plain sight, thus stealing a paper wallet is enough to get access to the funds. To increase security the private key can be encrypted using [BIP-38](#) proposal.

While a WIF key begins with '5' an encrypted WIF begins with '6P' so it is easy to differentiate between them.



Paper Wallets and Encryption (2)

Paper Wallets were frequently used in the past as a cold storage mechanism. Nowadays, they are not recommended and most users will use hardware wallets for security. The reason being that it caused confusion to new users.

New users are thinking of paper wallets as an (offline) account. When they import the key to a, say, mobile wallet to spend from, they assume it will behave as an account. In reality, if you import and send funds from a mobile wallet the latter will send the change to another address of the mobile wallet's (not the paper wallet's). From then on the paper wallet is useless and the remaining funds are in the mobile wallet. This was confusing to users.

As a purely cold storage mechanism the idea of paper wallets is quite good if you understand the above. A mnemonic code backup written in paper for a backup could effectively be considered an offline wallet (or cold storage wallet) if you don't have it imported to any device.

Paper wallets and mnemonic code backups could be destroyed relatively easily if they are in 'paper' form, e.g. by fire or water or mice! To remedy this, 'steel' wallets are used; same idea but with steel instead of paper. There are several companies that offer customizable steel wallets.



Conclusions



Conclusions

- We explained what addresses are and how they are generated
- We differentiated between deterministic and non-deterministic wallets and explained what paper wallets and mnemonic codes are



Further Reading



Self-assessment exercises

- Setup a Python environment and go through the examples of the tutorial (BasicPythonBitcoin-tutorial.pdf)
- Describe possible outcomes of mistyping a Bitcoin address when trying to send some bitcoins.
- What are the disadvantages of deterministic wallets?
- Use a [vanity generator](#) to create some addresses
- Experiment with the generation of HD keys and mnemonic codes using an [online generator](#)
- Write a program that creates 10 random private keys together with their addresses (compressed)
- Write a function that creates a Bitcoin address given a public key
 - Use the pseudocode that we provided in the lectures as a basis
- Write a function that expects a WIF private key and a passphrase and encrypts it using BIP-38



Further Reading

Mastering Bitcoin (Ch.4), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc>

(Bitcoin keys and addresses)

Hierarchical Deterministic Wallets - BIP-32 and BIP-44

<https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>

<https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki>

Mnemonic Codes - BIP-39

<https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming

Bitcoin Scripting 1

Konstantinos Karasavvas



Objectives of Session

- Understand transactions in more detail
- Explore how transaction signing works

In this session we go deeper into what constitutes a transaction and different ways of how transactions can be signed.



Agenda

- Transactions
- Signatures
- Conclusions
- Self-assessment exercises and further reading



Section 1: Transactions



Transactions and Scripting

A transaction consists of 1+ inputs and 1+ outputs. When an output is spent it can never be used again. All the bitcoins are transferred elsewhere (to a recipient, back to yourself as change, etc.). Outputs that are available to be spent are called *Unspent Transaction Outputs (UTXO)* and Bitcoin nodes keep track of the complete *UTXO set*.

When a UTXO is created we also specify the conditions under which this output can be spent. When you specify an input (the UTXO of a previous transaction) to spend you have to prove that you satisfy the conditions set by the UTXO.

The conditions and the proof that authorizes transfer are not fixed. A scripting language is used to define them. When a new output is created a script is placed in the UTXO called ***scriptPubKey*** or more informally *locking script*.

When we want to spend that UTXO we create a new transaction with an input that references the UTXO that we wish to spend together with an *unlocking script* or more formally a ***scriptSig***.



Transaction Types

The scripting language used in Bitcoin consists of several operations. Each operation is specified in hexadecimal by an opcode. It is a simple stack-based language that uses reverse polish notation (e.g. 2 3 +) that does not contain *potentially dangerous* programming constructs, like loops; it is a domain-specific language.

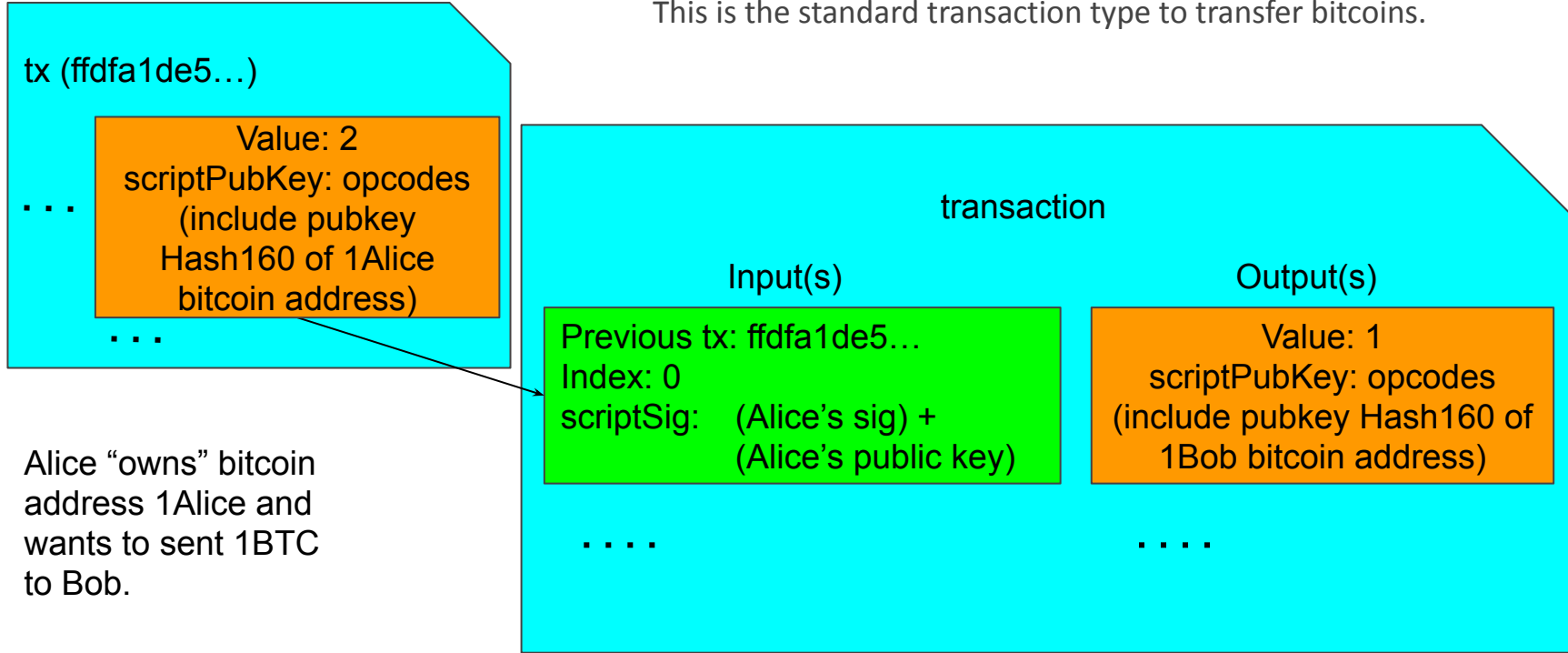
The most common transaction output type offering a standard way of transferring bitcoins around is *Pay-To-Public-Key-Hash* (P2PKH), which is effectively “pay to a Bitcoin address”. It is also possible, and used in the past, to pay directly to a public key with type *Pay-To-Public-Key* (P2PK).

Another transaction output type, *Pay-To-Script-Hash* (P2SH), allows for more complex scripts to be created.



P2PKH (1)

This is the standard transaction type to transfer bitcoins.



Alice “owns” bitcoin address 1Alice and wants to sent 1BTC to Bob.

P2PKH (2)

The locking script (`scriptPubKey`) looks like this:

```
OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG
```

The public key hash (*PKHash*) can be derived from the Bitcoin address that we want to transfer coins to by Base58Decoding the address and then removing the version and the checksum bytes.

The unlocking script (`scriptSig`) looks like this:

```
<Signature> <PublicKey>
```

The signature is the ECDSA signature of the hash of part of Tx that we create.

The validation to spend a UTXO consists of running the script of **scriptSig** plus **scriptPubKey**. Both scripts are added in the stack and executed as one script.



P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

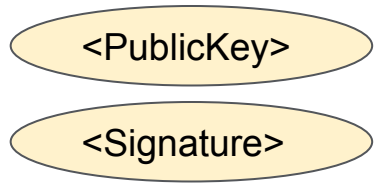
STACK:



P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

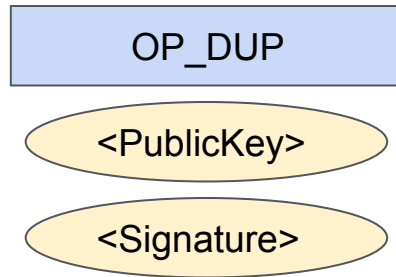
STACK:



P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

Duplicates the top stack item



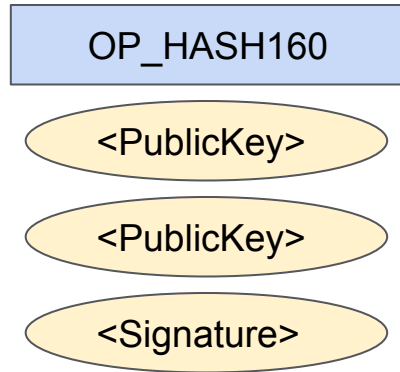
STACK:



P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

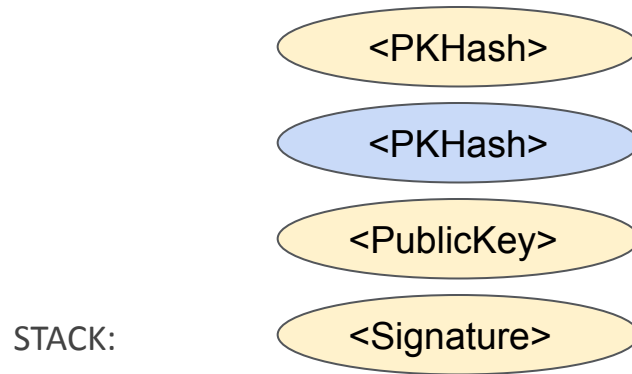
RIPEMD(SHA256(top item))



STACK:

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`



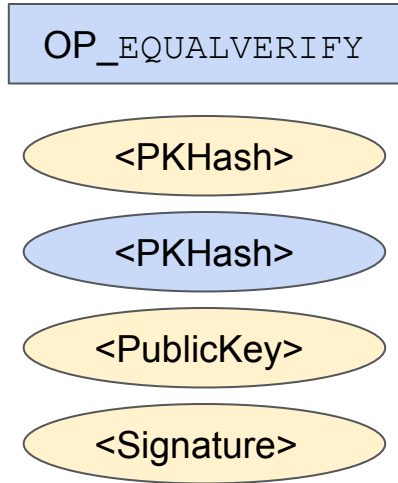
P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

OP_EQUAL checks 2 top items and replaces them with true or false

OP_VERIFY checks top item and if true removes it and if false it terminates the script

STACK:



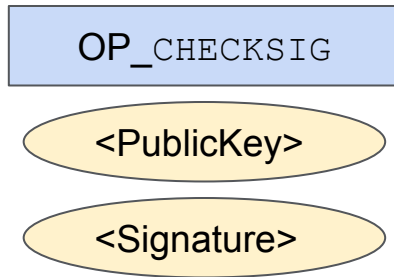
If the two `<PKHash>`es are equal the `<PublicKey>` provided was correct!

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

OP_CHECKSIG consumes a public key and a signature. The Tx that we wish to spend is hashed and then the system validates the signature using the verified public key.

STACK:



Signatures Reminder:
The sender hashes parts of the Tx (the message) and signs it.

The system then gets the Tx as well (the message) and hashes it. It then verifies that the signature and public key produce the same hash !

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

Success !

The system validated the ownership of the UTXO which is about to be spent.

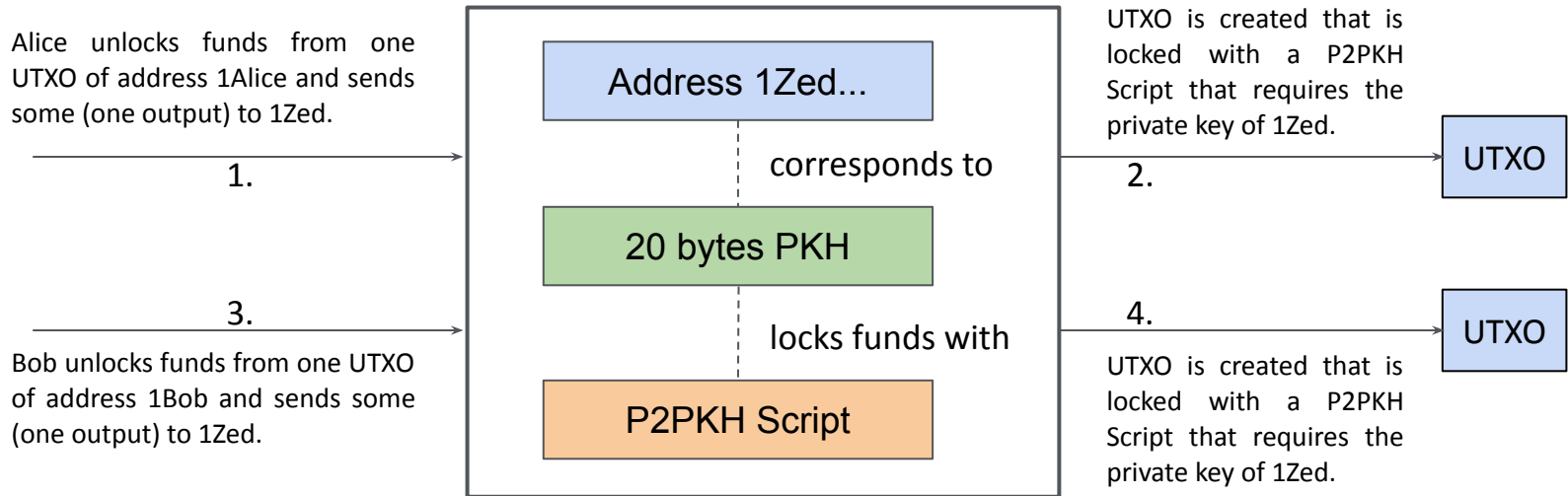
STACK:



true

Addresses, Locking Scripts and UTXOs (again!)

To help clarify how addresses, locking scripts and UTXOs relate look at the following diagram. Addresses 1Zed, 1Alice and 1Bob are short for the actual bitcoin addresses of Zed, Alice and Bob respectively. The diagram emphasises what happens when funds are sent to an address.



Example Transaction (1)

All transactions and blocks are available to examine. Let us examine the following mainnet transaction: 6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b

It has one input and one output.

Transaction View information about a bitcoin transaction

6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b

1Dj8woELooyfmsYH2ifPhP6vsqVTTpcaxJ



1524uZcTYKiECrKkYZwvzDoxRtc1mU23sM

1.2946 BTC

6 Confirmations

1.2946 BTC

Summary

Size	191 (bytes)
Received Time	2017-07-06 08:50:58
Included In Blocks	474484 (2017-07-06 08:53:29 + 3 minutes)
Confirmations	6 Confirmations
Relayed by IP	176.58.96.62 (whois)

Inputs and Outputs

Total Input	1.2957 BTC
Total Output	1.2946 BTC
Fees	0.0011 BTC
Fee per byte	575.916 sat/B
Estimated BTC Transacted	1.2946 BTC

Example Transaction (2)

Transactions, at the network layer, are transferred as a sequence of bytes according to a specific structure. We can see the byte sequence of this transaction by querying a mainnet node.

```
$ ./bitcoin-cli getrawtransaction
6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b
01000000014655ecb69a2660ee381b5e2d8e616c97bde4144d9bf3aeb2b514428414
7e1ba9000000006a47304402200b113ac8ff3699aa213055e3dcacea8509b7ffa36d
2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14e10f7f7fa806a88cfe7bba20
360deef9e74229a1d562f50121027f922a3403503d143404d2cf18df94899070673b
4cdee3e08be3c8db7e6467aaffffffff012067b707000000001976a9142c142e0bc0
1f9cc4623f6b4613696d5c98b1141e88ac00000000
```



Example Transaction (3)

```
./bitcoin-cli decoderawtransaction 01000000014655ecb ... 41e88ac000000000
{
  "txid": "6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b",
  "hash": "6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b",
  "size": 191,
  "vsize": 191,
  "version": 1,
  "locktime": 0,
```



Example Transaction (3)

```
"vin": [  
  {  
    "txid": "a91b7e14844214b5b2aef39b4d14e4bd976c618e2d5e1b38ee60269ab6ec5546",  
    "vout": 0,  
    "scriptSig": {  
      "asm":  
"304402200b113ac8ff3699aa213055e3dcacea8509b7ffa36d2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14e1  
0f7f7fa806a88cfe7bba20360deef9e74229a1d562f5[ALL]  
027f922a3403503d143404d2cf18df94899070673b4cdee3e08be3c8db7e6467aa",  
      "hex":  
"47304402200b113ac8ff3699aa213055e3dcacea8509b7ffa36d2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14  
e10f7f7fa806a88cfe7bba20360deef9e74229a1d562f50121027f922a3403503d143404d2cf18df94899070673b4cdee  
3e08be3c8db7e6467aa"  
    },  
    "sequence": 4294967295  
  }  
],
```

Example Transaction (3)

```
"vout": [  
  {  
    "value": 1.29460000,  
    "n": 0,  
    "scriptPubKey": {  
      "asm": "OP_DUP OP_HASH160 2c142e0bc01f9cc4623f6b4613696d5c98b1141e OP_EQUALVERIFY  
OP_CHECKSIG",  
      "hex": "76a9142c142e0bc01f9cc4623f6b4613696d5c98b1141e88ac",  
      "reqSigs": 1,  
      "type": "pubkeyhash",  
      "addresses": [  
        "1524uZcTYKiECrKkYZwvzDoxRtc1mU23sM"  
      ]  
    }  
  }  
]
```

Section 2: Signatures



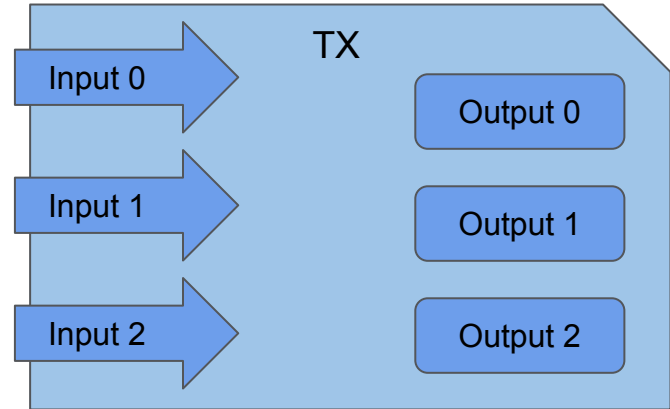
Signatures

When we create a new transaction we need to provide a signature for each UTXO that we want to spent. The signature proves:

- that the signer is the owner of the private key
- the proof of authorization is undeniable
- the parts of the tx that were signed cannot be modified after it has been signed

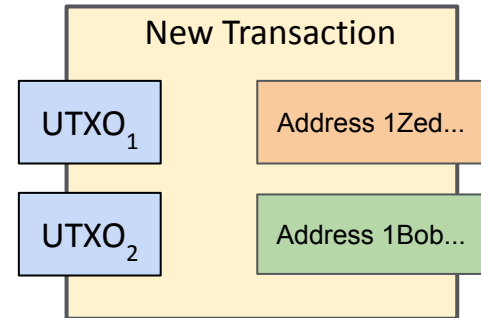
The digital signature algorithm used is ECDSA and each signature is serialized using **DER**. At the end of each signature we define its signature hash type (SIGHASH) by a 1 byte suffix.

The SIGHASH type determines which parts of the transaction are signed.



Signing Bitcoin Inputs

- Each transaction input has to be signed separately
- The message to be signed is the transaction itself with:
 - all other inputs scriptSigs should be empty
 - the input's scriptSig (the one that we sign) should be set to the scriptPubKey of the UTXO that we are trying to spend
 - follow additional SIGHASH rules



SIGHASH flags

Flag	Value	Description
ALL	0x01	Signs all the inputs and outputs, protecting everything except the signature scripts against modification.
NONE	0x02	Signs all of the inputs* but none of the outputs, allowing anyone to change where the satoshis are going.
SINGLE	0x03	Signs all the inputs* and only one output, the one corresponding to this input (the output with the same output index number as this input), ensuring nobody can change your part of the transaction but allowing other signers to change their part of the transaction. The corresponding output must exist.

* The sequence numbers of other inputs are not included in the signature, and can be updated.

SIGHASH flags modifier: ANYONECANPAY

With multiple inputs, each signature hash type can sign different parts of the transaction. If a 2-input transaction has one input signed with NONE and the other with ALL, the ALL signer can choose where to spend the funds without consulting the NONE signer.

Flag	Value	Description
ALL ANYONECANPAY	0x81	Signs all of the outputs but only this one input, and it also allows anyone to add or remove other inputs, so anyone can contribute additional satoshis but they cannot change how many satoshis are sent nor where they go.
NONE ANYONECANPAY	0x82	Signs only this one input and allows anyone to add or remove other inputs or outputs, so anyone who gets a copy of this input can spend it however they'd like.
SINGLE ANYONECANPAY	0x83	Signs this one input and its corresponding output. Allows anyone to add or remove other inputs.

Conclusions



Conclusions

- We explained how transactions really work in detail, including how transaction signing happens to protect parts of a transactions



Further Reading



Further Reading

Mastering Bitcoin (Chapters 6-7), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc>

Bitcoin's Developer Guide

<https://bitcoin.org/en/developer-guide>


Signature Hash Types

<https://bitcoin.org/en/developer-guide#signature-hash-types>





UNIVERSITY *of*
NICOSIA



BLOC-521 Digital Currency Programming

Bitcoin Scripting 2

Konstantinos Karasavvas



Objectives of Session

- Explore how scripting is done in Bitcoin
- Examine several ways to create new transactions

In this session we go deeper into how scripting is used to lock bitcoins and later unlock them to spend them. We then provide several examples on how to create transactions by calling a node's API or programmatically.



Agenda

- Scripting
- Creating Transactions
- Conclusions
- Self-assessment exercises and further reading



Section 1: Scripting



Script

Script is a simple scripting language used as Bitcoin's transaction processing language. The *friendly* notation we saw previously is the assembly equivalent. This is compiled to a byte sequence.

There are several [assembly opcodes](#) most of which are not used in the majority of scripts. However, the scripting language has a lot more potential than the applications that we see available.

There are constants, like OP_3 and OP_8 that represent 3 and 8 respectively. There are a range of arithmetic operations available, like OP_ADD or OP_SUB, boolean operators, cryptographic operators and others.

We have already seen the script that scriptPubKey and scriptSig need to contain in the typical P2PKH transaction type.



Script: Example 1

Script: OP_3 OP_4 OP_ADD OP_5 OP_SUB

Hex: 53 54 93 55 94

Stack: []
[3]
[3, 4]
[7]
[7, 5]
[2]

Result is: 2



Script: Example 2

Script: OP_3 OP_4 OP_EQUAL OP_IF OP_5 OP_ELSE OP_10 OP_ENDIF

Hex: 53 54 87 63 55 67 60 68

Stack: []
[3]
[3, 4]
[0]
[10]

Result is: 10



Script: Example 3

Just for completeness the hex sequence for a scriptPubKey of a P2PKH is:

```
scriptSig:      <Signature>          <PublicKey>
Hex:           49 11..ff 01          41 11..ff
scriptPubKey:  OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG
Hex:           76          a9          14 11..ff          88          ac
```

Notice that values are *pushed* into the stack with OPs that are implied (not visible in the script).



Script: More examples

What would unlock the following scriptPubKey's ?

OP_TRUE

OP_FALSE

OP_NOT

OP_NOP

OP_RETURN <20 bytes in hex>

OP_ADD OP_SUB OP_2 OP_EQUAL

OP_HASH256 <32 bytes in hex> OP_EQUAL

2 <pubkey A> <pubkey B> <pubkey C> 3 OP_CHECKMULTISIG



Debugging Script

To practice with scripts you can use a [Bitcoin Script IDE](#) or a command-line tool called, [btcdeb](#). Install the latter locally and you will have `btcc` and `btcdeb` available.

```
$ ./btcc OP_1 OP_2 OP_ADD
515293

$ ./btcdeb '[OP_1 OP_2 OP_ADD]'
btcdeb -- type `./btcdeb -h` for start up options
valid script
3 op script loaded. type `help` for usage information
script | stack
-----+-----
1      |
2      |
OP_ADD |
#0000 1
btcdeb> step
```



More OP codes: Arithmetic and Boolean

OP_1ADD (0x8b)	— Increment by one
OP_1SUB (0x8c)	— Decrement by one
OP_NEGATE (0x8f)	— Flip the sign of the number
OP_ABS (0x90)	— Make the number positive
OP_NOT (0x91)	— Flips 1 and 0, else 0
OP_ADD (0x93)	— Add two numbers
OP_SUB (0x94)	— Subtract two numbers
OP_MIN (0xa3)	— Return the smaller of two numbers
OP_MAX (0xa4)	— Return the larger of two numbers
OP_BOOLAND (0x9a)	— 1 if both numbers are not 0, else 0
OP_BOOLOR (0x9b)	— 1 if either number is not 0, else 0



More OP codes: Equality and Stack

OP_NUMEQUAL (0x9c) — 1 if both numbers are equal, else 0

OP_LESSTHAN (0x9f) — 1 if first number is less than second, else 0

OP_GREATERTHAN (0xa0) — 1 if first number is greater than second, else 0

OP_LESSTHANOREQUAL (0xa1) — 1 if first number is less than or equal to second, else 0

OP_GREATERTHANOREQUAL (0xa2) — 1 if first number is greater than or equal to second, else 0

OP_WITHIN (0xa5) — 1 if a number is in the range of two other numbers

OP_DEPTH (0x74) — Pushes the size of the stack

OP_PICK (0x79) — Duplicates the nth stack item as the top of the stack

OP_ROLL (0x7a) — Moves the nth stack item to the top of the stack

OP_SWAP (0x7c) — Swaps the top two stack items



More OP codes: Cryptographic and Conditional

OP_RIPEMD160 (0xa6)	— RIPEMD-160
OP_SHA1 (0xa7)	— SHA-1
OP_SHA256 (0xa8)	— SHA-256
OP_HASH160 (0xa9)	— SHA-256 + RIPEMD-160
OP_HASH256 (0xaa)	— SHA-256 + SHA-256
OP_CHECKSIG (0xac)	— Check a signature
OP_CHECKMULTISIG (0xae)	— Check a m-of-n multisig
OP_IF (0x63)	— If top stack item true execute block (up to OP_ELSE, if there)
OP_ELSE (0x67)	— If OP_IF top stack item is false executes OP_ELSE block
OP_ENDIF (0x68)	— Ends a if/else block



Section 2: Pay to Script Hash (P2SH)



P2SH (1)

P2SH is a type of transaction output ([BIP-16](#)) that moves the responsibility for supplying the conditions to redeem a transaction (locking script) from the sender of the funds to the redeemer (receiver).

Consider the scenario where we accept funds in an address that is not controlled by one person. For example it is typical for companies to allow spending from corporate accounts only if, say, 2 people agree. These are called multi-signature accounts. A multi-signature account requires M-of-N signatures in order to spend the funds. An address' locking script could enforce that. For example a 2-of-3 multi-signature locking script would look like:

```
2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG
```

If the company wanted to receive money in an address that multiple participants are needed to redeem it, it would not suffice to send just the address to the customers but also the locking script. This is impractical as a use case, has privacy implications and is also not efficient since the whole script would be recorded on the blockchain for every transaction.

P2SH (2)

A P2SH moves the responsibility for supplying the conditions to redeem a transaction (locking script) from the sender of the funds to the redeemer (receiver).

The locking script of such a transaction is quite simple:

```
OP_HASH160 [20-byte-hash-value] OP_EQUAL
```

The 20-byte hash is the hash of the redeem script:

```
RIPEMD-160( SHA-256( 2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG ) )
```

Using this hash we create a Bitcoin address (same process but instead of `OP_HASH160(pubkey)` we use `OP_HASH160(redeem script)`) using the version prefix of `0x05` that creates addresses that start with `3`.

We then disseminate only this address to the company's customers to send the funds.



P2SH (3)

When the company needs to spend the funds it would send the following unlocking script:

```
<Director's signature> <CFO's signature> <2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG >
```

Validation occurs in 2 steps. First we confirm that the redeem script equals the hash in the locking script:

```
<Director's signature> <CFO's signature> <2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG> OP_HASH160  
[20-byte-hash-value] OP_EQUAL
```

And then just validating the redeem script:

```
<Director's signature> <CFO's signature> 2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG
```

Transaction Output Types

- P2PK (TX_PUBKEY)
- P2PKH (TX_PUBKEYHASH)
- P2SH (TX_SCRIPTHASH)
- P2WPKH (TX_WITNESS_VO_KEYHASH)
- P2WSH (TX_WITNESS_VO_SCRIPTHASH)
- OP_RETURN (TX_NULL_DATA)
- Multisignature (TX_MULTISIG)
- Non-standard (TX_NONSTANDARD)

Non-standard transaction outputs are rejected (but not invalid) and not relayed by nodes. However, they can be mined if it is arranged with a miner.



Section 3: Creating Transactions



Using bitcoin-cli -- Node's API, high-level

```
./bitcoin-cli listunspent 0
[
  {
    "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
    "vout": 0,
    "address": "n1jnmQCyt9DHR3BYKzdbmXWM8M5UvH9nMW",
    "account": "",
    "scriptPubKey": "76a914ddcf9faf5625d6a96790710bbcef98af9a8719e388ac",
    "amount": 1.30000000,
    "confirmations": 0,
    "spendable": true,
    "solvable": true
  }
  ...
]

./bitcoin-cli sendtoaddress mnB6gSoVfUAPu6MhKkAfgsjPFBWmEEmFr3 0.1
```

The wallet chooses which UTXOs will be spent and in which order.



Using bitcoin-cli -- Node's API, low-level (1)

```
./bitcoin-cli createrawtransaction ''
> [
>   {
>     "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
>     "vout": 0
>   }
> ]
> '' ''
> {
>   "mqazutWCSnuYqEpLBznke2ooGimyCtwCh8": 0.2
> }''
01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b30000000
000fffffffff01002d3101000000001976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac
00000000
```

Create a transaction while specifying the UTXO to be spent.



Using bitcoin-cli -- Node's API, low-level (3)

```
"vout": [
  {
    "value": 0.20000000,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 6e751b60fcb566418c6b9f68bfa51438aefbe094 OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "mqazutWCSnuYqEpLBznke2ooGimyCtwCh8"
      ]
    }
  }
]
```



Using bitcoin-cli -- Node's API, low-level (4)

```
./bitcoin-cli signrawtransactionwithwallet
01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000
000fffffffff01002d3101000000001976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac
00000000
{
  "hex":
  "01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000
  006a4730440220404082ecae0b088e07647a5a4eb5c71626e001cbca9353bb6f7e6b212f0f95d0022
  02cdadf64f31b11e1901134abe7917d74105953aa983db099504891696277b86d01210306a6ae64fb
  b424a81260a6c47f3cb52eec39c4b40ded8b05e150458b95ea6465fffffffff01002d310100000001
  976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac00000000",
  "complete": true
}

./bitcoin-cli decoderawtransaction 01000000014039..8ac0000000
...
./bitcoin-cli sendrawtransaction 01000000014039..8ac0000000
error code: -26, error message:, 256: absurdly-high-fee
```



Using HTTP JSON-RPC (1)

JSON-RPC is a simple protocol that specifies how to communicate with remote procedure calls using JSON as the format. It can be used with several transport protocols but most typically it is used over HTTP.

A user name and password has to be provided in `bitcoin.conf`. By default only local connections are allowed, but other connections can be allowed for **trusted** IPs with `rpccallowip` configuration option.

```
rpcuser=kostas  
rpcpassword=too_long_to_guess
```

Using HTTP JSON-RPC (2)

```
$ curl --user kostas --data-binary '{"jsonrpc": "1.0", "id": "curltest", "method": "getinfo", "params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:18332/
Enter host password for user 'kostas':

{
  "result":
  {
    "version": 130100,
    "protocolversion": 130000,
    "balance": 2.27500000,
    ...
  }
  "error": null,
  "id": "curltest"
}
```

Note that `getinfo` is not available from v0.16. Instead use `getblockchaininfo`, `getnetworkinginfo`, `getmininginfo` and `getwalletinfo`.



Using a library

A Python library that wraps Bitcoin's API calls is `python-bitcoinrpc`. Install with pip and try it out.

```
from bitcoinrpc.authproxy import AuthServiceProxy, JSONRPCException

# user and pw are rpcuser and rpcpassword respectively
user = "kostas"
pw = "too_long_to_guess"      # bad practice !!
rpc_connection = AuthServiceProxy("http://%s:%s@127.0.0.1:18332"%(user, pw))
block_count = rpc_connection.getblockcount()
print(block_count)
```

All API calls can be used, including the ones to create a transaction with either `sendtoaddress` or `createrawtransaction + signrawtransaction + sendrawtransaction`.



Using another library (1/2)

Another Python library that wraps Bitcoin's API and is faithful to the programming types that the original C++ library has is `python-bitcoinlib`. Install with pip and try it out.

```
import bitcoin.rpc
from bitcoin import SelectParams
from bitcoin.core import COIN, b2lx
from bitcoin.wallet import CBitcoinAddress

SelectParams('testnet')

rpc = bitcoin.rpc.Proxy()
addr = CBitcoinAddress('mwtaAhm3Fdjnc525kMENrpP7zsqE8VvWdZ')

txid = rpc.sendtoaddress(addr, 0.001 * COIN)
print(b2lx(txid))
```

Note that the result of `sendtoaddress` is a txid which is bytes so we have to convert to hex before we display it (note that transactions are displayed in reversed or little-endian).



Using another library (2/2)

Using the same library we can construct the transaction from scratch! Please consult this [example](#) for a P2PKH transaction.

- What does this example do?
- What is need to complete the transaction?
- Can you identify a problem with that example?

And [here](#) (or [here](#)) is another example for a P2SH transaction that contains the obsolete P2PK transaction in the redeem script (`<pubKey> OP_CHECKSIG`).



Using yet another library

Using the [python-bitcoin-utils](#) (`pip install bitcoin-utils`) library we can also construct the transaction from scratch! Please consult this [example](#) for a P2PKH transaction.

There is also example code for:

- Creating [P2PKH with different SIGHASH](#)
- Creating a [P2SH address](#) and how to [spend it](#)
- How to [create](#) and [spend](#) a non-standard tx
- Using a [proxy](#) to make calls directly to a Bitcoin node

Please note that the raw hexadecimal of the tx is created and this needs to be send to a node using a proxy, either using the library or from the command line with `sendrawtransaction`.



Conclusions



Conclusions

- We explained how scripting is used to lock/unlock funds
- Through examples we demonstrated how Bitcoin's Script language works
- Learned how to create transactions with the Bitcoin API and via libraries



Further Reading



Self-assessment exercises

1. Write a program that creates and spends a transaction.
 - a. Allow the user to select which UTXOs to be used
2. In mainnet, how can we estimate what is an appropriate fee to include to a transaction?
 - a. Using tools
 - b. Programmatically
3. Write a scriptPubKey script that requires both a key and password to unlock.
4. The Bitcoin white paper (PDF) is stored on the blockchain. The transaction id is: 54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713. Can you extract the data and reconstruct the PDF?

You are welcome to use the forums to report issues, questions or your thoughts in general!



Further Reading

Mastering Bitcoin (Chapters 6-7), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc>

Bitcoin's Developer Guide

<https://bitcoin.org/en/developer-guide>

Scripting Language and all opcodes

<https://en.bitcoin.it/wiki/Script>





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming

Bitcoin Scripting 3

Konstantinos Karasavvas



Objectives of Session

- Examine more advanced scripts
- Examine several ways to create more sophisticated transactions

In this session we explain more sophisticated scripts to lock funds as well as explain the segwit upgrade and what it entails.



Agenda

- Segregated Witness (Segwit)
- P2MS (Multisignature outputs)
- Data Storage
- Conclusions
- Self-assessment exercises and further reading

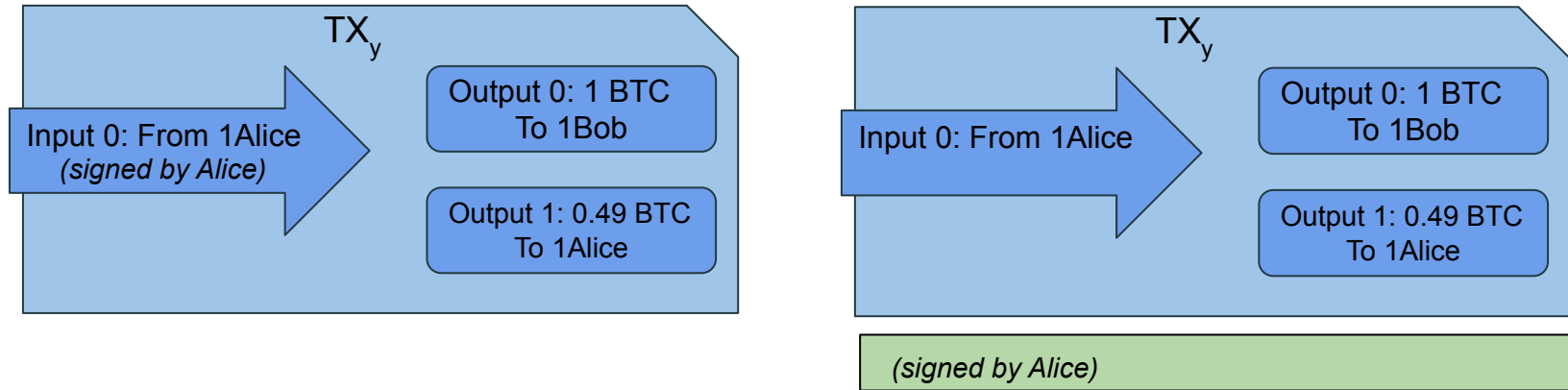


Section 1: Segregated Witness (SegWit)



Segregated Witness (SegWit)

Segregated Witness is a consensus change that introduces an update on how transactions are constructed. In particular it separates (segregates) the signatures or unlocking script (witness); a transaction input does not contain an unlocking script anymore and the latter is found in another structure that goes alongside the transaction.



The segwit upgrade is described in detail in BIPs [141](#), [143](#), [144](#) and [145](#) and provides several [benefits](#). We will examine two of them here: *transaction malleability* and *effective block size increase*.

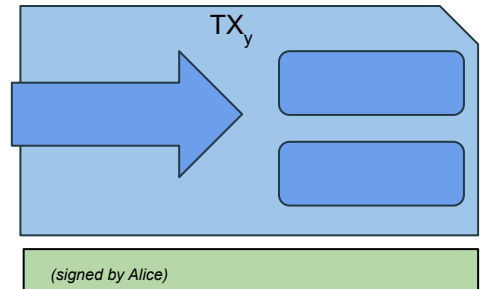
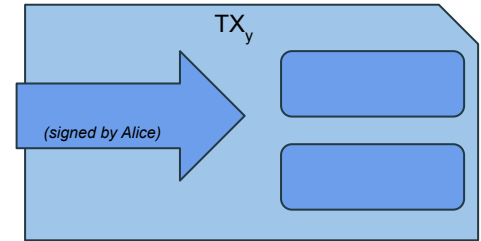
Transaction malleability

Each transaction is uniquely identified by its transaction identifier or txid. The txid is constructed by hashing the serialized transaction (the blue part).

It is possible to slightly change the unlocking script, e.g. by a miner, so that the resulting transaction is semantically identical to the original, thus still valid. This can be accomplished, for example, by slightly changing the encoding format of the signature.

That is a problem because given how the txid is created even the slightest modification will change the txid. While the transaction is identical, i.e. funds will be moved exactly as intended, our ability to monitor this transaction is problematic given that we will be checking for confirmations in a txid that is not valid anymore.

With segwit inputs, however, the unlocking script (the green part) is not part of the txid construction and thus it is impossible to modify it. A non-malleable txid is more reliable and allows for more advanced scripts/solutions like the lightning network.



Effective Block Size Increase

The actual block size remains the same, at 1MB. However, the unlocking scripts are now not part of the block and thus more transactions can fit into the 1MB limit.

Segwit introduces the concept of block *weight*, a new metric for the size of blocks. A block can have a maximum weight of 4MBs. The non-witness part bytes of a transaction are now multiplied by 4 to get its weight while the witness part bytes are multiplied by 1, a discount of 75%. This allows for an effective maximum block size increase of $\sim 1.8x$, if all transactions use segwit.

The *virtual* size, or `vsize` of a transaction is the size in bytes of a transaction including the segwit discount. For non-segwit transactions `size` and `vsize` are identical.



SegWit Transaction Output Types

Segwit introduces two new transaction types, *Pay-to-Witness-Public-Key-Hash (P2WPKH)* and *Pay-to-Witness-Script-Hash (P2WSH)*, which are the segwit equivalent of P2PKH and P2SH respectively. They are sometimes called *native segwit* to differentiate from *nested segwit*.

The locking script (`scriptPubKey`) of these new types consists of two elements, a version byte and the witness program. The version byte introduces versioning in the witness program of the script. That is another benefit of segwit, since it allows for easy updates based on a new version.

Remember, that when signing to spend any output we need to provide the locking script, which is used to substitute the `scriptPubKey` before we calculate the transaction digest and sign it. For segwit transaction types each witness program corresponds to a predefined template script that is called *scriptCode*.

For example the `scriptCode` for a P2WPKH output is:

```
OP_DUP OP_HASH160 <pubkey-hash> OP_EQUALVERIFY OP_CHECKSIG
```

where the `pubkey-hash` is substituted with the witness program. Note that this is used for calculating the digest and not for validation (see next slide).



Native Witness Program - P2WPKH

In segwit version 0, a P2WPKH witness program is just the 20-byte public key hash. The unlocking script (scriptSig) should be empty and the witness structure contains the unlocking script.

```
scriptPubKey: 0 6b85f9a17492c691c1d861cc1c722ff683b27f5a
scriptSig: ""
witness: <signature> <pubkey>
```

Validation:

1. The '0' in scriptPubKey specifies that the following is a version 0 witness program.
2. The length of the witness program (20-bytes) indicates that it is a P2WPKH type.
3. The witness must consist of exactly two items
4. The HASH160 of the <pubkey> must match the 20-bytes witness program
5. Finally, the signature is verified by: <signature> <pubkey> CHECKSIG



Native Witness Program - P2WSH

In segwit version 0, a P2WSH witness program is just the 32-byte script hash. The unlocking script (scriptSig) should be empty and the witness structure contains the unlocking script as well as the witness program script.

```
scriptPubKey: 0
6b85f9a17492c691c1d861cc1c722f92c691c1fa17492c691c1d861683b27f5a
scriptSig: ""
witness: 0 <signature1> <1 <pubkey1> <pubkey2> 2 CHECKMULTISIG >
```

Validation:

1. The '0' in scriptPubKey specifies that the following is a version 0 witness program.
2. The length of the witness program (32-bytes) indicates that it is a P2WSH type.
3. The witness must consist of an input stack followed by a serialized script (witness script)
4. The SHA256 of the witness script must match the 32-bytes witness program
5. Finally, the witness script is deserialized and executed after the remaining witness stack:
0 <signature1> 1 <pubkey1> <pubkey2> 2 CHECKMULTISIG



P2SH Witness Program

It is possible for a non-segwit aware wallet to pay to a segwit address by embedding P2WPKH or P2WSH into a P2SH. The recipient will provide a P2SH address to the sender who can send funds as usual. The recipient can then use the redeem script which is actually the witness script to spend the funds.

P2SH(P2WPKH): Get the hash of the P2WPKH scriptPubKey and use it in P2SH as usual:

```
HASH160 <0 6b85f9a17492c691c1d861cc1c722ff683b27f5a>
```

```
HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL
```

P2SH(P2WSH): Get the hash of the P2WSH scriptPubKey and use it in P2SH as usual:

```
HASH160 <0 3b892c61cc15f9a17492c691c1d86a17492c61cc1c722ff683b27f5a>
```

```
HASH160 b3adef9719ec8659f4b2b0b3e0547268b3b19288 EQUAL
```



Segwit and soft-fork

Changing the transaction format is normally a hard-fork. The new transactions would not be accepted by the nodes running the old software. To go around that and implement the new functionality as a soft-fork additional effort was required. Without going into too much details:

- The original transaction format was not changed. The **scriptSig** would just be empty and the **witnesses** would go in a new structure.
- The header *should* represent the whole block and thus a witness merkle root is calculated (from all transactions' witness scripts) and included in an OP_RETURN output (explained in detail later) of the coinbase transaction. Being in the coinbase means that the witnesses are represented in the header via the `hashMerkleRoot`.
- Witness data are provided only when nodes ask for them and thus old nodes will get blocks without the witness data, i.e. new nodes will remove witness data before relaying the blocks to old nodes. To old nodes, segwit blocks would look like blocks that contain some non-standard transactions.
- Old nodes trying to spend a segwit output would violate the clean stack rule
 - `OP_0 <bytes in hex>` will remain in the stack.



Section 2: P2MS (Multisignature outputs)



P2MS

Pay to multi-signature or *Pay-to-Multisig* is a standard output type that was introduced before P2SH. Its aim was to provide a way for bitcoins to be locked by several public keys which could belong to different people. Typically, only a subset of signatures would be required. For example for a 2-of-3 multisig would require at least 2 signatures from 2 of the corresponding private keys.

In the previous session we have seen an example of a 2-of-3 multisig wrapped in P2SH. This is the typical way to use multisig after P2SH was created because P2MS has several drawbacks:

- It is limited to 3 public keys while P2SH allows up to 15.
- It has no address format. To send funds to a P2MS the sender needs to know the multisig script.
- The public keys are visible even before an output is spent.

To construct and lock funds in a P2MS output we use the exact script that we described for P2SH but lock the funds there directly.

Note that the `CHECKMULTISIG` opcode has a bug where it pops an extra element from the stack. For backward compatibility the bug cannot be fixed and thus to avoid the issue we add an additional dummy value at the beginning of the unlocking script. Typically, `OP_0` is used as a dummy but anything is valid.



Section 3: Data Storage



Data Storage - Indirectly (1/3)

The blockchain ensures that all existing entries are tamper-proof resistant; modifications and deletions are not allowed. This makes it quite useful for *permanently* storing data that will stand the test of time, which is ideal for certain applications like notary services, certificate ownership and others.

However, Bitcoin's blockchain was not designed for storage in general and data could only be stored indirectly. Examples include adding data to coinbase transactions, to transaction outputs and to multi-signature addresses.

Coinbase transactions have no typical inputs. Instead they have a special coinbase field that holds arbitrary data up to 100 bytes. This field is used for extra nonce space but anything can be stored. For example* in the genesis block we have:

```
PUSHDATA (04)                                     FFFF001D
PUSHDATA (01)                                     04
PUSHDATA (45)  5468652054696D65732030332F4A616E2F32303039204368616E63656C6C6F72206
                F6E206272696E6B206F66207365636F6E64206261696C6F757420666F722062616E
                6B73
```

'The Times 03/Jan/2009 Chancellor on brink of second bailout for banks'

* example from Ken Shirriff's Blog



Data Storage - Indirectly (2/3)

Note that only miners can store data to a coinbase field. An alternative for typical users would be to store data in the outputs themselves in fake addresses. Remember that the output address is represented as the public key hash of 20 bytes (40 hex characters). Those can be faked to represent the data that we need to store.

The satoshis sent to such a fake address will be lost forever since there is no (known) private key that corresponds to it. In the past, when the value of bitcoin was small it was easy to *afford* to store this way.

For example*, fake address “15gHNr4TCKmhHDEG31L2XFNvnpnEcnPSQvd” corresponds to hex “334E656C736F6E2D4D616E64656C612E6A70673F” that is stored in the blockchain and if it is converted to Unicode you have “3Nelson-Mandela.jpg?” which is the filename of an image of Nelson Mandela that follows!

Storing data this way creates an overhead for the UTXO set in every node in the network. These addresses will never be used (i.e. the satoshis there are lost) but the system is not aware so they need to keep track of them as unspent outputs.

* example from Ken Shirriff’s Blog



Data Storage - Indirectly (3/3)

Similarly multi-signature addresses could be used to store data in a similar fashion as with fake addresses. The script of a multi-sig contains all the public keys which can be faked to include arbitrary hex data.

```
1
PUSHDATA (65) [204e00005cfccf40377abcaf271c0003c5f21904146227000000000026000000000000069469d04002392
44d42fe726b29b083ec863bb48f5c1765c23e5cf23f5]
PUSHDATA (65) [19204be9a427ced5afc27b4b9f80cf936c5cbfe298cd12b69edd0de82ed5baa8819ac53d0a003f68ac0db0
6562018288c011dfd4f0ce04e0b151e5c722c6e8b74e]
PUSHDATA (65) [9203dd005bca4ad7d701b935181ab33060f8e7a6471d3a1210490fae16e88a0401c5011d6cfece87599be8
ad82609e432b25c5b6c9c7942bb830e0fb919ae71481]
3 CHECKMULTISIG
```

For example* the wikileaks cablegate data, a 2.5MB file (cablegate-201012041811.7z) stored using 130 separate transactions. Each transaction was donating 1 satoshi to wikileaks.

As with fake addresses storing with this method also spams the UTXO set.

* example from Ken Shirriff's Blog

Data Storage - Directly

Storing data on the blockchain with the above methods was frowned upon the community because of the overhead it was placing on the running nodes. Others argued that as long as the transaction fee is paid there is no reason why it should be considered spam. The compromise was the introduction of an operator (OP_RETURN) specifically dealing with storing small amount of data on the blockchain.

The OP_RETURN was followed by a maximum of 80 bytes of data. No satoshis were required to be sent (other than the transaction fees) and more importantly, OP_RETURN was not *bloating* the UTXO set.

Since 80 bytes is a very small amount of data it is usually used to store a hash (or *digest* or *digital fingerprint*) of some data ensuring the integrity of the data rather than the immutable existence of the data itself. Alternatively, it can be used to encode meta-protocol information, such as used by [Counterparty](#), the [OMNI protocol](#), [Colored Coins](#) and University of Nicosia's [BDIP protocol](#).

Example of OP_RETURN:

```
OP_RETURN PDATA(20) 4f1edef24e9e2a169f56e1b08bac361f30ae936d32232652dc51be1860ecd714
6a      20      4f1edef24e9e2a169f56e1b08bac361f30ae936d32232652dc51be1860ecd714
```



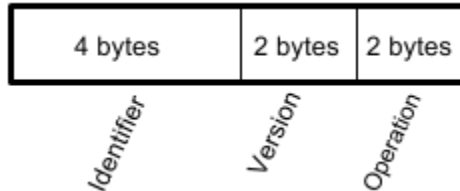
Example Data Storage - BDIP meta-protocol (1/2)

The *Blockchain Document Issuing Protocol* meta-protocol allows anchorage of files into the blockchain. The files' integrity can then be verified independently the issuer, i.e. consulting only the blockchain. It contains several operations that store appropriate data on the blockchain. While we will not elaborate on the developed platform we will briefly illustrate how it was encoded (for more information check [here](#)).

Operations:

OP_ISSUE (0x0004), OP_REVOKE_BATCH (0x0008), OP_REVOKE_CREDS (0x000c), ...

Header:

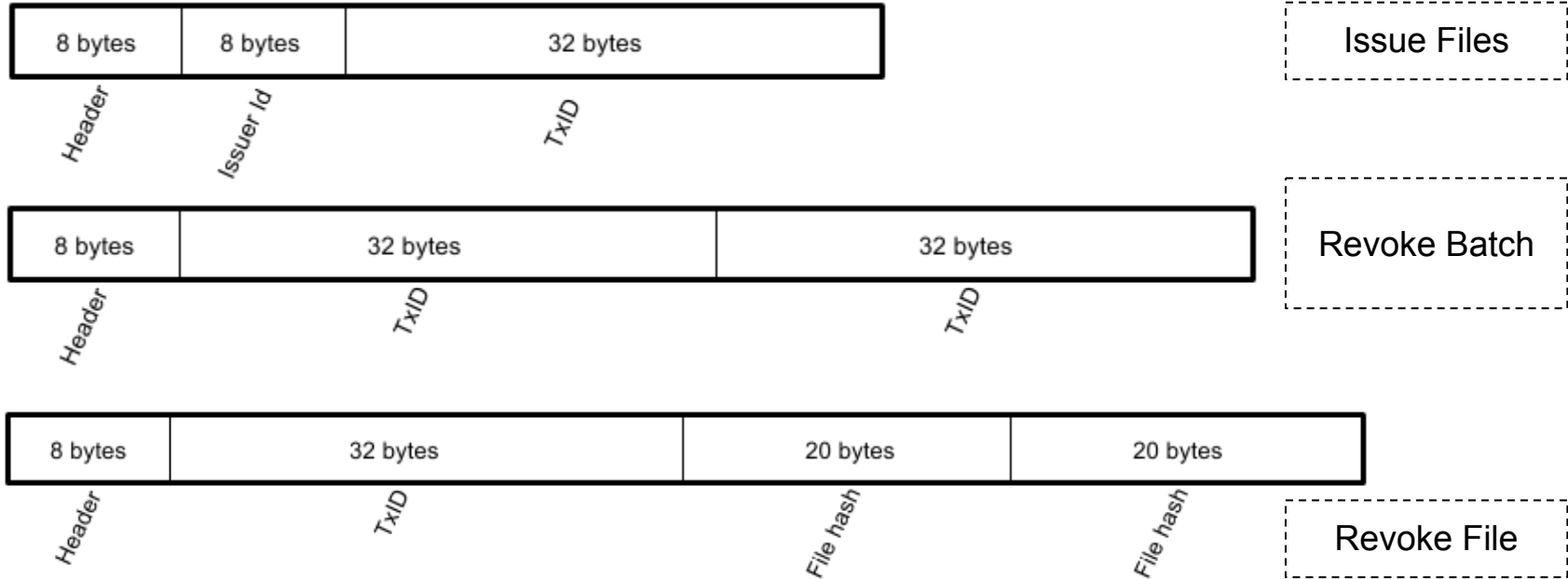


Identifier: 43524544 (“CRED”)

Version: 0001

Operation: 0004

Data Storage - BDIP meta-protocol (2/2)



Github: <https://github.com/verifiable-pdfs/blockchain-certificates>

Conclusions



Conclusions

- We explained what the segwit upgrade is, what changed and the consequences of that change
- We explained how data can be stored in the blockchain



Further Reading



Self-assessment exercises

1. Explain why using segwit is expected to have an effective block size increase of x1.8.
2. Write a program that uses OP_RETURN to store a string on the blockchain (testnet).
3. Create a script the implements a simple 1-of-2 multisignature scheme.

You are welcome to use the forums to report issues, questions or your thoughts in general!



Further Reading

Mastering Bitcoin (Chapters 6-7), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc>

Bitcoin's Developer Guide

<https://bitcoin.org/en/developer-guide>

Scripting Language and all opcodes

<https://en.bitcoin.it/wiki/Script>

Ken Shirriff's Blog article regarding stored data in Bitcoin's blockchain

<http://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.html>

Bitcoin wiki: OP_RETURN

https://en.bitcoin.it/wiki/OP_RETURN





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming

Bitcoin Scripting 4

Konstantinos Karasavvas



Objectives of Session

- Examine more advanced scripts
- Examine several ways to create more sophisticated transactions

In this session we explain more sophisticated scripts to lock funds as well as explain the taproot upgrade and what it entails.



Agenda

- Timelocks
- RBF & CFPF
- Schnorr and Taproot
- Conclusions
- Self-assessment exercises and further reading



Section 1: Timelocks



Timelocks - Absolute time (1/5)

Timelocks is a mechanism for postdating transactions or to lock funds for specific periods of time. It applies only to version 2 transactions. There are two different types of locking, one for absolute and one for relative time. In each one we can specify timelocks at transaction level or at script level.

This feature was part of the initial Bitcoin implementation. Every transaction can include a timelock (nLocktime) to specify the earliest time that a transaction may be added to a block. Wallets were setting this value to '0' meaning that the transaction is valid anytime. Later on, a soft-fork allowed to specify the time in terms of the block height. Possible values:

Value	Meaning
0	Transaction is always valid.
< 500 million	Specifies the earliest block height that this transactions can be added.
>= 500 million	Specifies the block header time (Unix Epoch) after which the transaction can be added to a block.



Timelocks - Absolute time (2/5)

Absolute nLocktime is used in some wallets to prevent *fee sniping*. Fee sniping is a theoretical attack that involves large miners/pools mining two (or possibly more) blocks in an attempt to reorganize past blocks. The miner can then add the highest-fee transactions from the previously valid blocks plus any high-fee transactions in the mempool.

The Bitcoin Core wallet (from 0.11.0) creates transactions that include an nLocktime of the current best height plus one. Thus, the transaction is valid for the next block as normal but in the case of a re-org a miner cannot add this transaction in a previous block. This means that, if all transactions use this mechanism, the miner will not be able to gain any new fees by including new transactions to older blocks.

$$\begin{aligned} \text{nLocktime} &= \text{current_best_height} + 1 \\ \text{nSequence}^* &= 0xFFFFFFFFE \end{aligned}$$

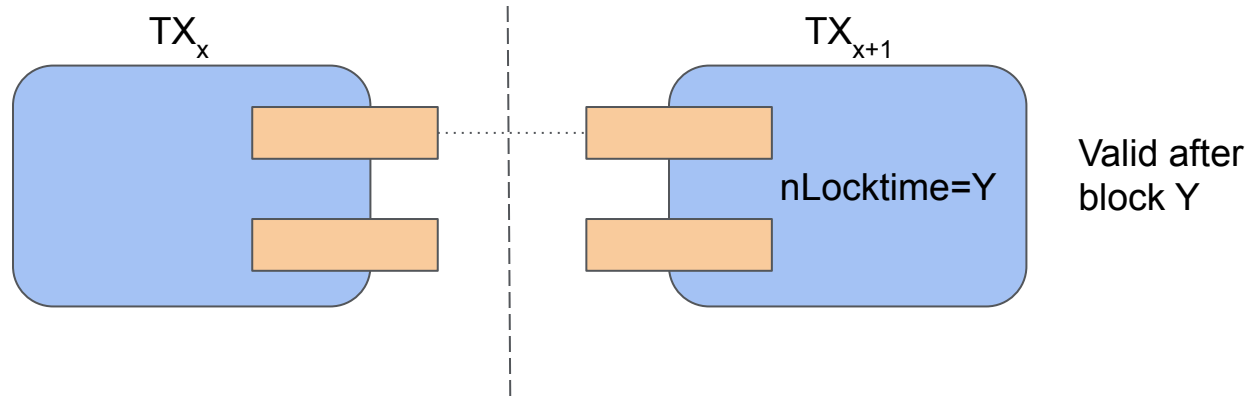
This will be more important as the miners' reward is reduced further making transaction fees the major source of income for miners.

* To enable timelocks. nSequence was intended for another use but it was never implemented and is now used to specify an active timelock. Typical transactions have nSequence of 0xFFFFFFFF.



Timelocks - Absolute time (3/5)

For example, if we want to spend a UTXO of transaction TX_x in block Y (a block in the future, say 600000) we need to create a transaction that spends it but also set `nLocktime=Y`. Then this new transaction TX_{x+1} will be invalid until that block height.



Timelocks - Absolute time (4/5)

Note that nLocktime creates a transaction that *cannot* be included in the blockchain until the specified block/time. This means that the person who created the transaction could create another transaction to spend the funds, invalidating the nLocktime transaction.

Absolute locktime is achieved at the script level using *CHECKLOCKTIMEVERIFY (CLTV)*. In late 2015 BIP-65 soft-fork redefined OP_NOP2 as OP_CHECKLOCKTIMEVERIFY allowing timelocks to be specified per transaction output. To spend the output, the signature and public key are required as usual but the nLocktime field of the spending transaction also needs to be set to an equal or greater value of CLTV's timelock value. If not the script will fail immediately.

A scriptPubKey example that locks an output until 'expiry_time':

```
<expiry_time>                OP_CHECKLOCKTIMEVERIFY                OP_DROP
OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG
```

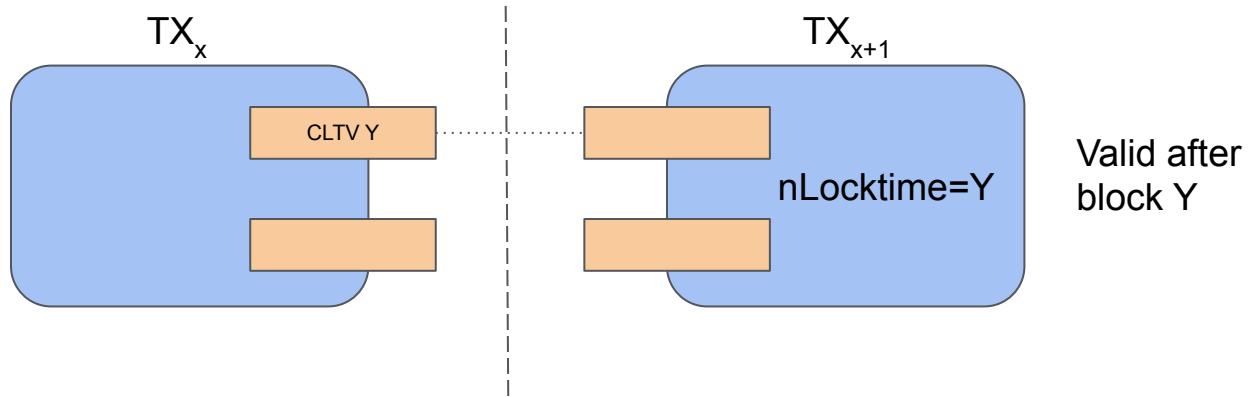
The nLocktime can be expressed in either block height or timestamp but it has to be the same type as the one used in the expiry time. Timelocks need to be activated by setting nSequence to 0xFFFFFFFF.

Since a script with CHECKLOCKTIMEVERIFY becomes part of the blockchain it cannot be invalidated as described above.



Timelocks - Absolute time (5/5)

For example, we can create a locking script with CLTV on block Y (a block in the future, say 600000) and send some funds to it (and keep sending). If we want to spend it we need to create a transaction that spends it but also set nLocktime to (at least) 600000. Then this new transaction TX_{x+1} will be invalid until that block height.



Spending a CLTV + nLocktime P2SH

Using `python-bitcoinlib` library. The locking script is just the timelock plus the equivalent script of a P2PKH. Key is the private key used to unlock the P2PKH. The funds are locked until block 200 (regtest) and they will be sent to 'mrAfr4SjMp2CRSgCYiPC88DZn4ULhBzXzF'.

```
...
txin_redeemScript = CScript([200, OP_CHECKLOCKTIMEVERIFY, OP_DROP,
                             OP_DUP, OP_HASH160, Hash160(key.pub),
                             OP_EQUALVERIFY, OP_CHECKSIG])

...
txid = 1x('ed38d54853c52c82cf0fc19e8c479898cf471a4d47297408a80ed9229402ed3a')
vout = 0
txin = CMutableTxIn(COutPoint(txid, vout), nSequence=0xffffffffe)
txout = CMutableTxOut(2*COIN, CBitcoinAddress('mrAfr4SjMp2CRSgCYiPC88DZn4ULhBzXzF')
                    .to_scriptPubKey())
tx = CMutableTransaction([txin], [txout], nLockTime=200)
sighash = SignatureHash(txin_redeemScript, tx, 0, SIGHASH_ALL)
sig = key.sign(sighash) + bytes([SIGHASH_ALL])
txin.scriptSig = CScript([sig, key.pub, txin_redeemScript])
...
```



Timelocks - Relative time (1/4)

Relative timelocks were introduced in mid-2016 with BIPs 68, 112 and 113 as a soft-fork that made use of the nSequence field of an input. The original idea behind nSequence was that a transaction in the mempool would be replaced by using the same input with a higher sequence value. This assumes that miners would prefer a higher sequence number transactions instead of a more profitable one... so it was never implemented.

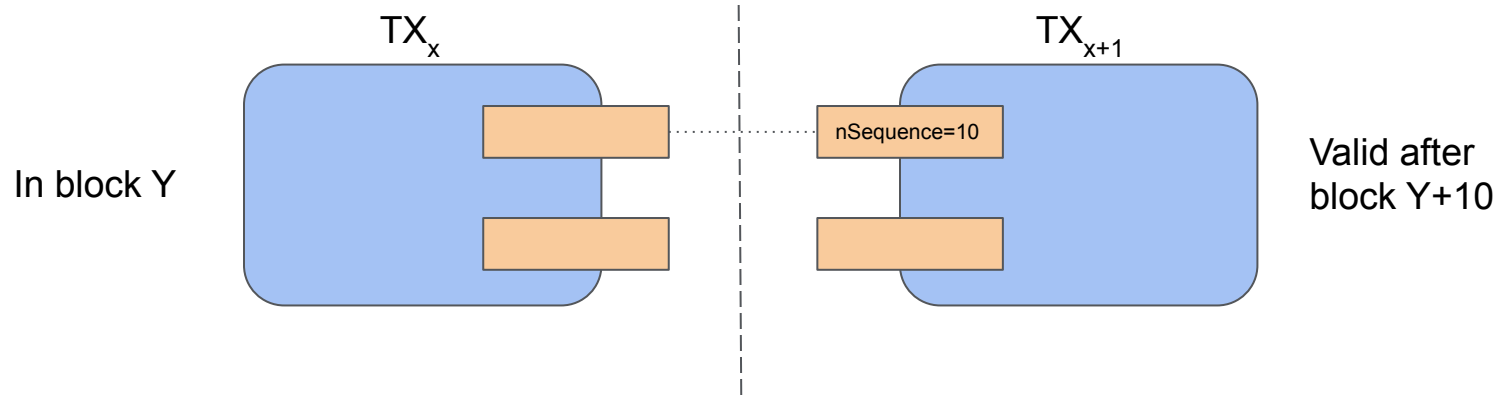
nSequence was repurposed (BIP-68) for relative timelocks. If the most significant bit of the nSequence (32 bit) field was 0 (i.e. 0x7FFFFFFF) then it was interpreted as a relative timelock. Then for timelocks bit 23 would specify the type (block height or Unix Epoch) and the last 16 bits the value.

Type (bit 23)	Meaning of last (least significant) 16 bits
0	The number of blocks that need to pass based on the height of the UTXO which the input spends.
1	The number of 512 seconds intervals that need to pass based on the timestamp of the UTXO which the input spends.



Timelocks - Relative time (2/4)

For example, if we want to spend a UTXO of transaction TX_x after 10 blocks we need to create a transaction that spends it but also set nSequence to 10. Then this new transaction TX_{x+1} will be invalid until TX_x gets 10 confirmations.



Timelocks - Relative time (3/4)

The script-level equivalent of relative timelocks is using *CHECKSEQUENCEVERIFY (CSV)* defined in BIP-112. It replaces `OP_NOP3` with `OP_CHECKSEQUENCEVERIFY`. When we create a transaction that spends a UTXO that contains a CSV, that input requires to have `nSequence` set with an equal or greater value to the CSV parameter value. Otherwise it will fail immediately.

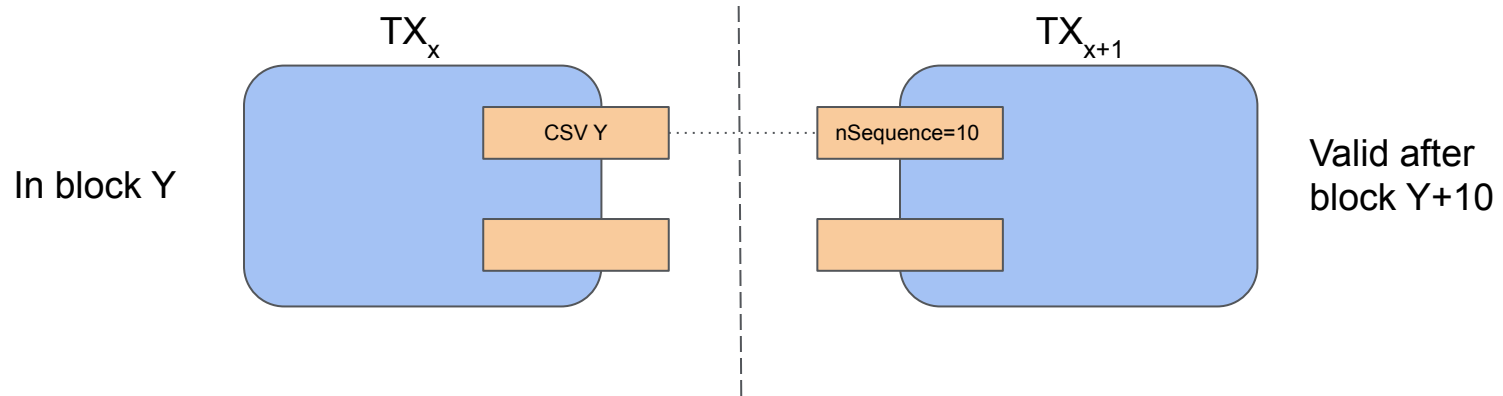
Expiry date can be expressed in either block height or timestamp (as previously discussed) but it has to be the same type as the one used in the `nSequence` field. Note that in the script only the block height or timestamp is included* and not the whole `nSequence` field.

* If timestamp the 23 bit has to exist and be set. Also note that integers in the script should be serialized as signed integers in little-endian.



Timelocks - Relative time (4/4)

For example, we can create a locking script with CSV with a value of 10 blocks and send some funds to it (and keep sending). If we want to spend it we need to create a transaction that spends it but also set the nSequence of the input that spends it to 10. Then this new transaction TX_{x+1} will be invalid until TX_x gets 10 confirmations.



Timelocks - Summary

Type	Location	Time specification	In Blockchain	Example
nLocktime	Transaction	Absolute	No	Similar to a will. Your heirs could get the funds in ~2040 but you could spend them (change will) in between.
nLocktime + CLTV	Script	Absolute	Yes	Lock funds as part of a deal that allows no one access until ~1-Jan-2020. Also CLTV-based payment channels.
nSequence	Input	Relative	No	Lock funds as part of a deal that prohibits the other party to spend funds until ~3 months have passed but you can.
nSequence + CSV	Script	Relative	Yes	Lock funds as part of a deal that allows no one access until ~3 months have passed. Payment channels, Lightning network



Section 2: RBF and CPFP



Replace-By-Fee (RBF)

RBF (specified in [BIP-125](#)) is a mechanism for replacing a transaction that is still in the mempool. It is primarily useful for re-sending a transaction of yours in case it was stuck, e.g. due to low fees. However, it may be useful for other [reasons](#).

It applies only to version 2 transactions. You need to set nSequence to a value of 0x01 upto 0xffffffff-2. However, since that such a value could also enable relative timelocks one has to be careful. Typically, for RBF you set the nSequence value to 1, which makes relative timelocks irrelevant, or from 0xf0000000 to 0xffffffffd which disables relative timelocks.

A replaceable transaction has the “bip125-replaceable” flag set to “yes” in its JSON display. Setting the bitcoin node option `-walletreplaceable` makes all transactions to be replaceable.

To make it work, in addition to setting the nSequence the transaction needs to reuse one or more of the same UTXOs and increase the fees (consult the BIP for more details).

There is an easy way to RBF a transaction (that is, of course, replaceable) by using bumpfee:

```
$ bitcoin-cli -named bumpfee txid=53fe...ffb4
```



Child-Pays-For-Parent (CPFP)

Child-pays-for-parent or CPFP is a mechanism (or trick, if you wish) for including a previous transaction (parent) in a block by creating a transaction (child) that spends one of the UTXOs of the parent. Miners will notice that the new transaction uses another one and will consider both transactions' fees when deciding whether to include it in the next block.

For example if someone sends you bitcoins but the transaction was stuck (e.g. due to low fees) you as a recipient can create a transaction that tries to spend the bitcoins from your address from the unconfirmed transaction. The fees of this transaction should be quite high to properly incentivize the miner (e.g. proper fees for two transactions!).

If the fee is high enough the miner will want to include the new (child) transaction and in doing so he is forced to include the initial transaction (parent) in the same block.

Can a sender of a transaction use CPFP to include it to a block?

Does it makes sense to do so?



Section 3: Schnorr and Taproot



Schnorr Signatures

Schnorr Signatures ([BIP-340](#)) is another signature scheme that is compatible with ECDSA's secp256k1 that Bitcoin uses. It is introduced with the Taproot upgrade and it will be valid only in Segwit v1 output scripts. All other transaction output types will be unaffected by the change and keep on using ECDSA.

Schnorr signatures improve over ECDSA in both privacy, scalability and efficiency. First off, they are 64 bytes long instead about 71 bytes that ECDSA has. That is immediately an 11% gain per signature which saves space in the block itself as well as disk space.

Secondly, they have the mathematical property of allowing signature aggregation; several signatures can be combined into one. This can make multi-signature outputs particularly efficient since a single signature would be required irrespective of the m-of-m scheme. Moreover, validation is faster since a single signature is validated. One protocol that can be used to orchestrate public key and signature aggregation is Musig.

Finally, cross-input signature aggregation would also provide great benefits, i.e. if signatures of different inputs could be combined into a single signature. However, this is work in progress since it has additional complexities.



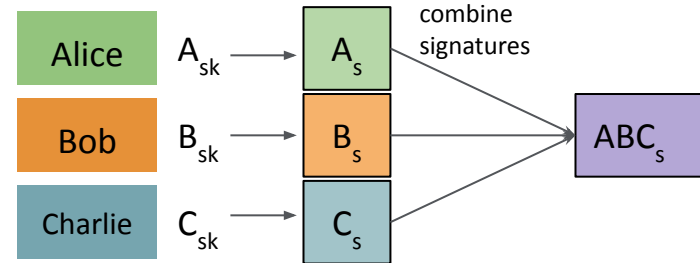
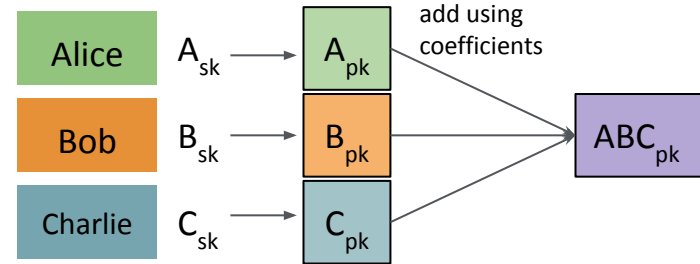
Musig

Musig is a protocol that enables public key and signature aggregation. With Musig users can combine their public keys to generate a single public key which is indistinguishable from any other public key.

This public key can then be shared to others to send funds.

When the time comes for the users to spend the funds they will use their secret keys (private keys) to create partial signatures. These, when aggregated will correspond to the aggregated public key.

Note that from the perspective of someone viewing the blockchain, it will look like a single signature verification.



Taproot

Taproot allows for different ways of spending an output. We can have two main spending paths. The default spending path is a single public key (or multi-party key using Musig, since it is indistinguishable). In addition, there is an alternative spending path, which can be a single script or multiple scripts. Only the specific script that is used is revealed during spending. Thus, privacy is improved in multi-party contracts.

Note, that if the default path is used there is no way for someone to know if there is an alternative path with several scripts. That is revealed only when an alternative script is spend. Generally, the default path would be the most common use case.

Taproot is also the most recent upgrade proposal for Bitcoin, which includes three BIPs:

- Schnorr Signatures ([BIP-340](#))
- a new SegWit v1 output type based on Taproot, Schnorr and Merkle branches ([BIP-341](#))
- Taproot script validation rules ([BIP-342](#))

Taproot

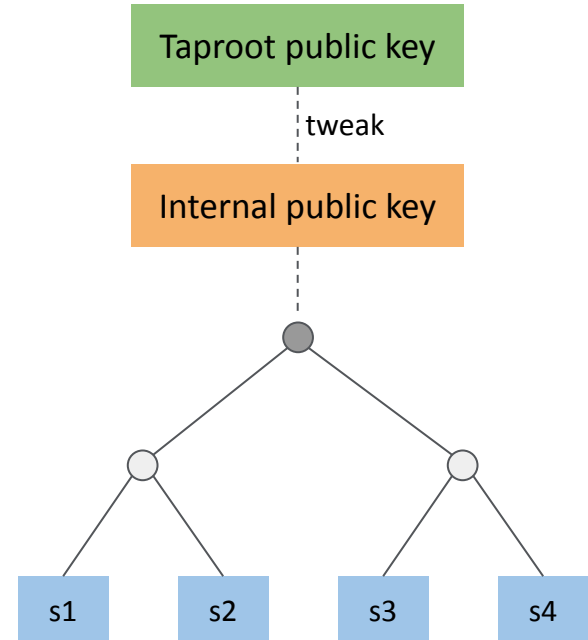
The taproot public key is constructed by the actual public key by tweaking it; typically just by EC adding a number (*commitment*) to the public key.

Then to spend from the taproot public key the private key needs to also be tweaked using the same number (*commitment*).

Then we can use the tweaked private key to sign the transaction.

Default spending path

Alternative spending path(s)
(OR conditions)



Tapscript

Tapscript is the Bitcoin Script that is allowed in Segwit v1 taproot outputs. Effectively it is a different version of Script where:

- Signature opcodes now validate Schnorr signatures
- Multisig opcodes replaced with `CHECKSIGADD` opcode
- Allows versioning (alternative path scripts could be of different version)
- Has several reserved opcodes to add new functionality later
 - Similar to NOP but will immediately succeed and return

New output descriptors are required to describe taproot outputs but these have not been finalized yet. For example: $\text{ts}(\text{pk}(\text{key}))$ could describe a taproot output with a single public key

The hash of a tapscript is the number (commitment) used to tweak the internal public key!

The simple case is for the alternative path to have a single tapscript. In that case to spend it we would need:

- the witness unlocking the tapscript
- the tapscript itself
- the internal public key



Taptree

We have already mentioned that there can be several alternative paths or tapscripts and not just one. This is accomplished by using a binary merkle tree with the leaf nodes being the tapscripts. When a merkle tree or taptree is used we tweak the internal public key with the merkle tree root.

To spend an alternative path of a taproot output with multiple scripts we need to provide:

- the witness unlocking the tapscript
- the tapscript itself
- the internal public key
- the inclusion proof that verifies that the hash of the specific script is part of the merkle root

Note that the idea of using a merkle tree was first in other proposals like [MAST](#) (Merkalized Abstract Syntax Trees) but it evolved as part of Taproot. With MAST, like P2SH, one would be able to differentiate between simple public key payments (e.g. P2PKH) and the pay to script payments from the address. Taproot solves this by using Schnorr aggregate signatures and the tweaking we previously mentioned.

Conclusions



Conclusions

- We learned how we can enhance our scripts to lock funds that include time constraints
- We explained what the taproot upgrade is, Schnorr signatures and the new taproot outputs



Further Reading



Self-assessment exercises

1. Write a program that creates any timelock (testnet)
2. Are relative timelocks at a transaction level propagated through the network?

You are welcome to use the forums to report issues, questions or your thoughts in general!



Further Reading

Mastering Bitcoin (Chapters 6-7), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc>

Bitcoin's Developer Guide

<https://bitcoin.org/en/developer-guide>

Scripting Language and all opcodes

<https://en.bitcoin.it/wiki/Script>

Bitcoin wiki: Timelock

<https://en.bitcoin.it/wiki/Timelock>

Schnorr Signatures and Taproot

https://www.youtube.com/watch?v=1gRCVLgkyAE&list=PLPrDsP88ifOVTEJf_jQGunDUS05M9GdIC&index=1

<https://hackernoon.com/excited-for-schnorr-signatures-a00ee467fc5f>

<https://bitcoinops.org/en/topics/musig/>





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming P2P Network and Forking Konstantinos Karasavvas



Objectives of Session

- Overview of P2P network
- SPV and Bloom Filters
- Discuss blockchain forking and what it means
- Explain Soft-forks
- Explain Hard-forks

In this session we will look briefly introduce the P2P network and how SPV clients work. We also discuss what forking is and how the network manages alternative implementations and changes.



Agenda

- The P2P Network
- Simplified Payment Verification
- Forking
- Conclusions
- Self-assessment exercises and further reading



Section 1: P2P Network



P2P Network: Introduction

A Bitcoin full node serves several functions to the network.

- Routing Node; propagates transactions and blocks
- Full blockchain; also called archival node
- Wallet
- Miner

By default most nodes will have a wallet (irrespective of its use) and will be able to propagate information through the network. By default they are also archival nodes, i.e. they keep the complete list of all blocks from genesis. However, lately, some nodes opt to prune the size of the blockchain for storage purposes. Finally, only a few of those will provide mining services.

There are currently (mid 2020) more than 10400 nodes in the network with ~99% of them using the Bitcoin Core implementation while the rest consists of alternatives like Bcoin, Bitcoin Unlimited, Bitcoin Classic, etc. A significant number of nodes (~21%) are using the Tor network.



Discovery

When a node is run for the first time it needs to discover other peers so that it joins the P2P network. This is accomplished with several methods:

DNS seeds: A list of (hardcoded) DNS servers that return a random subset of bitcoin node addresses. It sends a `getaddr` message to those peers to get more bitcoin addresses and so forth. The peers reply with an `addr` message that contains the addresses. The node can be configured to use a specific DNS seed overriding the defaults by using the `-dnsseed` command-line option.

Seed nodes: A list of (hardcoded) node IP addresses from peers that are believed to be stable and trustworthy. This is a fallback to DNS seeds. A specific node can be specified by using the `-seednode` command-line option.

Node addresses are stored internally so the above discovery methods are only required at first run. From then on the stored addresses can be used to remain up-to-date with active nodes in the network.

A list of the connected peers can be acquired with the `getpeerinfo` command and a node can connect to specific (trusted) peers with the `connect` option.



Handshaking and synchronisation

When a node connects to a new peer it initiates a *handshake* by sending a `version` message to establish the compatibility between peers. If the receiving peer is compatible it will send a `verack` message followed by its own `version` message.

As previously discussed a `getaddr` message is send next expecting several `addr` messages in return.

Initially, a node that starts for the first time only contains the genesis block and will attempt to synchronise (Initial Block Download) the blockchain from its peers. It sends a `getblocks` message which contain its current best block as a parameter. The receiving peers reply with an `inv` (inventory) message that contains a maximum of 500 block hashes after the initiator's best block. The initiator can then `getdata` to request the blocks themselves. The receiver will reply with several `block` messages each containing a single block.



Block Propagation and Bitcoin Relay Network

The faster a miner receives a new block the faster they can start working on the next block. Network latency is extremely important and since the P2P network takes some time (at least for the needs of miner) there are specialized networks to help with block propagation, e.g. *Fast Internet Bitcoin Relay Engine (FIBRE)*.

FIBRE does not only help less-connected miners to compete with the bigger mining farms but more importantly reduces the chance that a solution will be propagated before another node finds a second solution, thus reducing forks and orphan/stale block rates.

Note that the sole purpose of a relay network is to help propagate blocks fast between interested parties (like merchants, miners). They do not replace the P2P network rather provide additional connectivity between some nodes.

<http://bitcoinfibre.org>

<https://www.falcon-net.org/>

Another technology to improve propagation is Compact blocks ([BIP-152](#)) which reduces the transaction's size contained in a block.



Section 2: Simplified Payment Verification



Simplified Payment Verification (1/2)

A full node is a node that downloads and validates all the blocks and their transactions from the genesis block up to the most recent block. This is very secure since for a node to be fooled an attacker has to provide an alternative *longer* blockchain which aggregates more hashrate than the current one. To accomplish this the attacker needs to commence a 51% attack (and sustain it!) which is economically nonsensical.

Although the most secure method it is impractical/infeasible for low-resource devices like mobile phones to be expected to fully validate and store all transactions. For this reason an alternative called a *light* or *thin* or *SPV* client (or wallet) was proposed. Such a client needs to download only the header* of each block (80 bytes) and then query full nodes for blocks regarding specific transactions as the need arises.

* Instead of synchronizing by asking entire blocks (`getblocks`) they ask only for the block headers (`getheaders`).



Simplified Payment Verification (2/2)

An SPV client requests information about transactions relevant to its keys from a full node. The full node notifies the SPV client which blocks contain the relevant transactions together with the merkle proof. Now the client knows the merkle root (from the headers), the transaction hash as well as the merkle proof, so it can validate that the transaction was indeed included in a specific block.

This does not make the transaction automatically secure. The *depth* of the block is really important. The more confirmations the more certain the SPV client can be that this is a final transaction.



SPV Issues

A full node cannot easily fool an SPV client that the transaction is in a block but it is easy to make the client believe that a transaction is *not* in a block. Effectively, this can be used as a form of DoS attack which can be significant combined with a sybil attack; the more dishonest nodes the easier to achieve this attack. In addition, it can also be used to double-spend against SPV nodes.

To remedy this an SPV client needs to be well connected to random nodes and have at least one honest node to talk to.

Another issue is that user privacy could be at stake if certain measures are not taken. For example, asking information for specific transactions (that include keys of interest) indirectly *tells* a full node of the public addresses corresponding to this user. This is partly mitigated by *bloom filters* defined in [BIP-37](#).



Bloom Filters (1/2)

Bloom filter, from wikipedia:

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set".

The SPV client uses a bit array with size N with all the bits set to zero. Then, it applies k hashing functions, which produce a number from 0 to $N-1$, to one of the transactions that it requires more information for. Each hashing will result to a number up to $N-1$ which will be set to 1. This will occur for every transaction that the SPV client requires more information; thus it is possible that some bit fields will be set to 1 from hashing several addresses, without knowing for certain which ones.

The bloom filter request to a node is accomplished with `filterload`.



Bloom Filters (2/2)

The full node will then run the same hashing functions to all its addresses and if an address's bit fields are 1 for all hashes it will include extra information for this address to the result. The result is a *merkle block*, which is the block header, the TxIDs of the requested transactions together with a partial merkle tree used to verify that the transactions are in the merkle root. The actual transactions are sent separately.

The larger the bit field size and the less the number of hashing functions will result to greater privacy at the expense of more bandwidth.

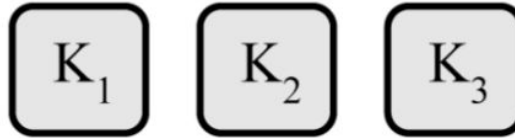
The SPV client can use `filteradd` to add new transactions to the existing filter and `filterclear` to remove it completely.



Bloom Filters - Example* (1/6)

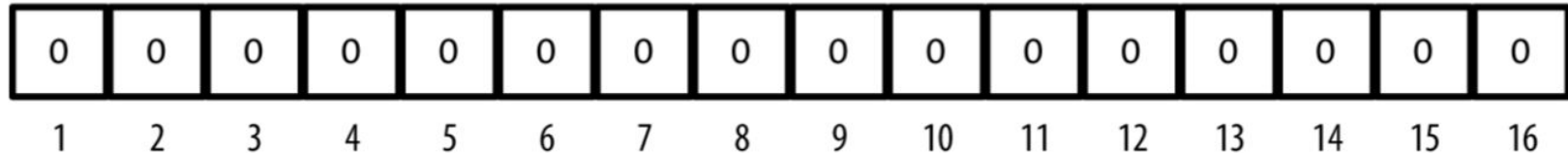
$k = 3$
 $N = 16$

3 Hash Functions



Hash Functions Output
1 to 16

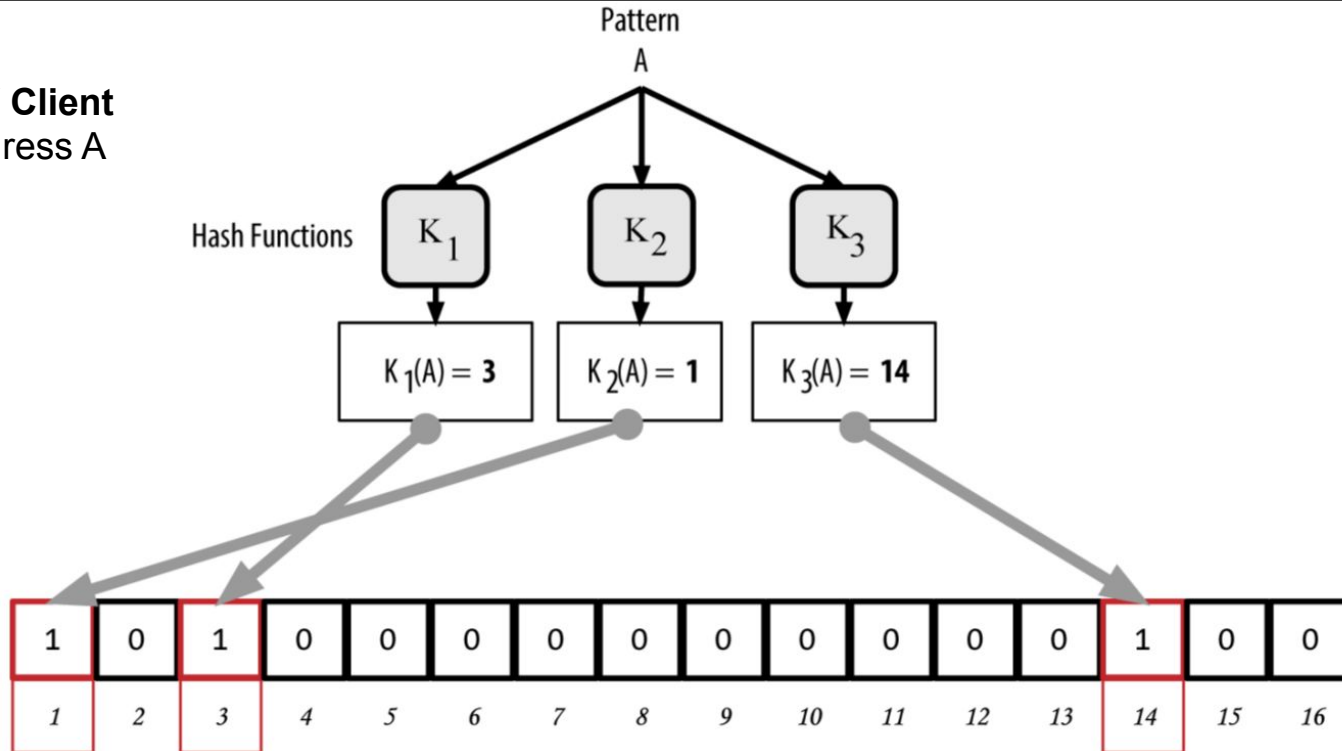
Empty Bloom Filter, 16 bit array



* Diagrams from [Mastering Bitcoin](#) book, Chapter 8

Bloom Filters - Example* (2/6)

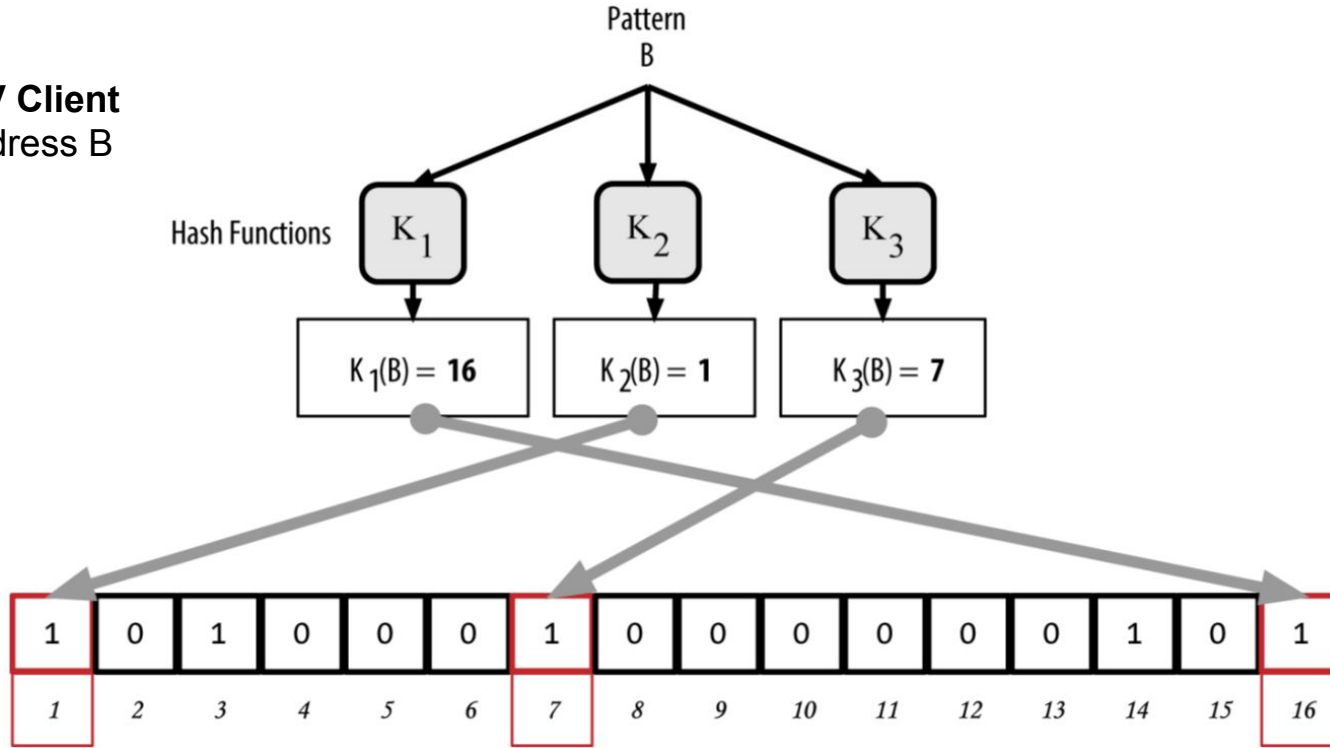
SPV Client
Address A



* Diagrams from [Mastering Bitcoin](#) book, Chapter 8

Bloom Filters - Example* (3/6)

SPV Client
Address B



* Diagrams from [Mastering Bitcoin](#) book, Chapter 8



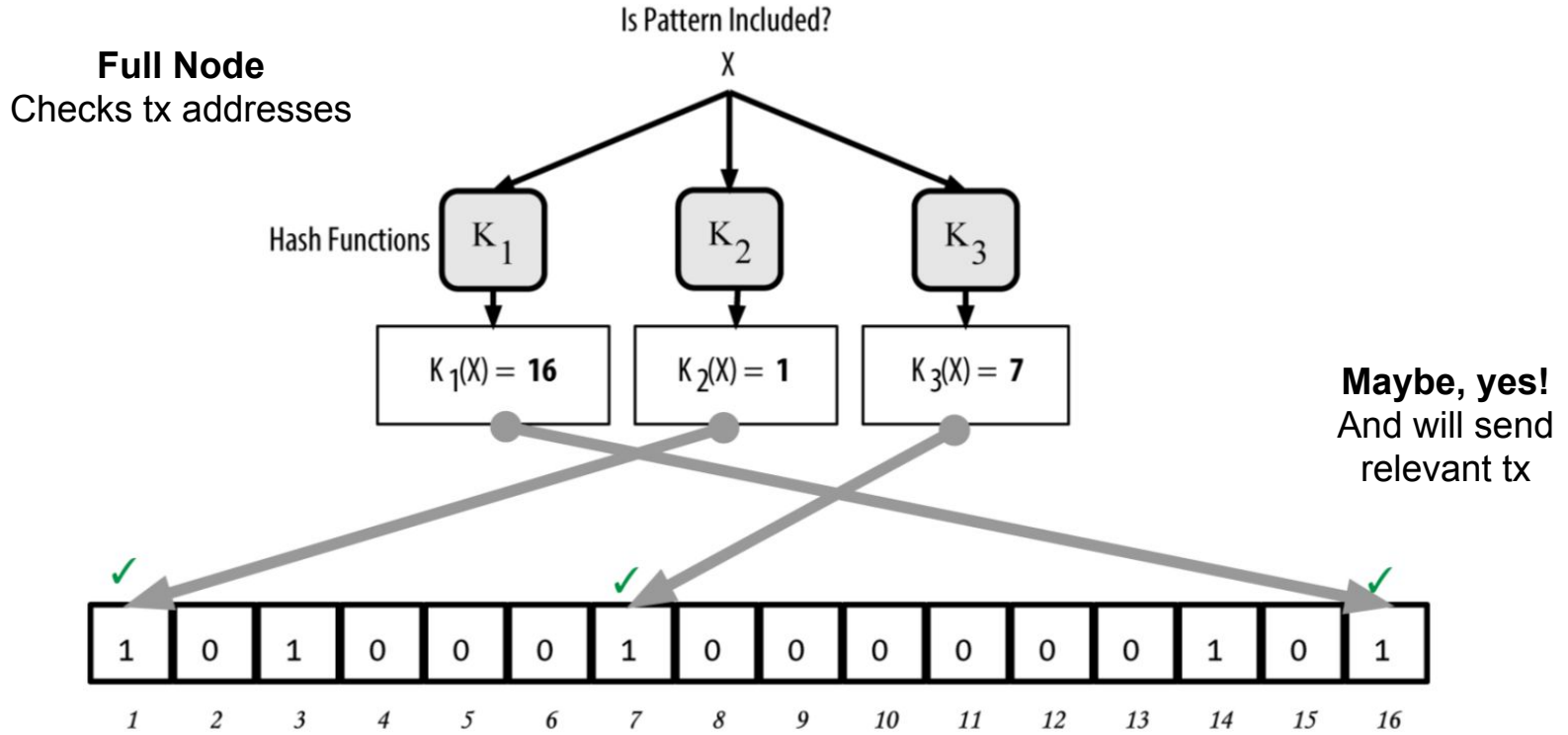
Bloom Filters - Example (4/6)

SPV Client sends the following:

1	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

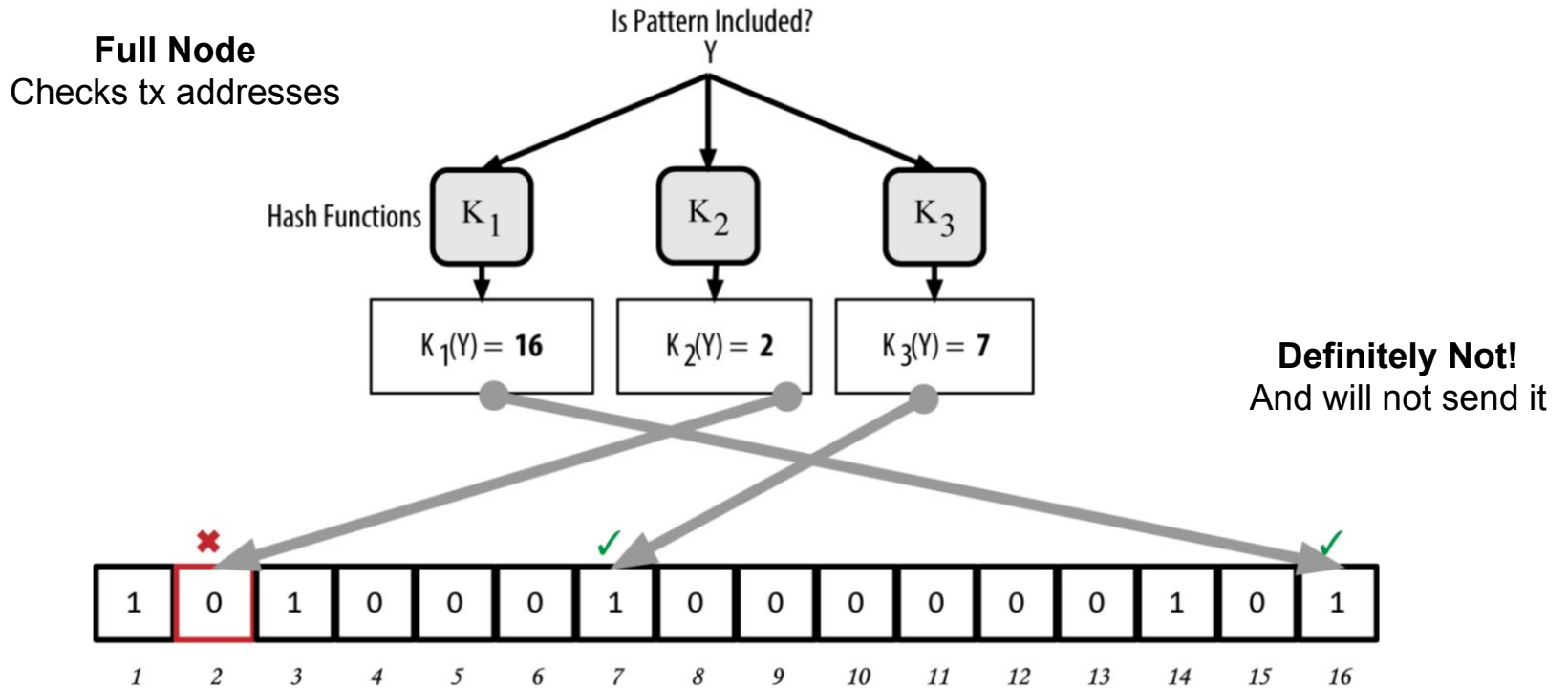


Bloom Filters - Example* (5/6)



* Diagrams from [Mastering Bitcoin](#) book, Chapter 8

Bloom Filters - Example* (6/6)



* Diagrams from [Mastering Bitcoin](#) book, Chapter 8

Section 3: Forking



Software Development Forks

A software project fork occurs when some developers take a copy of the project and develop it independently of the original. This is not just another development branch, this is a **divergence of direction**; effectively we now have two separate projects and the community splits accordingly.

Project forking is an important aspect of open source development allowing different opinions and roadmaps to become a reality. Some notable examples are:

- Linux Mint from Ubuntu (from Debian)
- MariaDB from MySQL
- PostgreSQL from Ingres
- OpenSSH from OSSH
- Inkscape from Sodipodi (from Gill)
- Plex from XBMC
- ...



Blockchain Forks (1)

A blockchain fork occurs when different peers on the network run code that implements incompatible rules. This can happen because of a software project fork when some developers take a copy of a blockchain project and develop it independently of the original but it could also happen due to a bug in a simple upgrade.

If the rules implemented change to a degree that the messages are not compatible with the original rules then some peers will start rejecting some of the messages with the possibility that the peer to peer network is effectively split into two networks, depending on the kind of change that occurred; i.e. the blockchain will fork and different peers will add blocks to different blockchains.

Note, that since running new code might result in a network (aka chain) fork the only way to update Bitcoin software to change protocol features is by forking.

Reminder: temporary branching on the blockchain is relatively common and it is part of the Nakamoto consensus. Forking refers to compatibility breaking changes between peers.



Blockchain Forks (2)

Forks can occur when nodes on the network run different versions of the software. This is the case when the Bitcoin software is being upgraded, e.g. from core v0.11.2 to core v0.12.0. This is a scheduled fork and if all peers agree on the change and upgrade the software in a timely manner there will be no issues.

Alternatively, competing versions might run, e.g. core v0.12.0 and classic v0.12.0. The two groups will have different roadmaps on where they wish Bitcoin to go and can compete to gain the majority of the hashing power in order for their blockchain to prevail.

We can have intentional forks due to software upgrades or alternative implementations and unintentional forks due to incompatibilities caused by bugs.

There are two different types of blockchain forking:

- Soft-forks; blocks that would be valid (to old nodes) are now invalid
- Hard-forks; blocks that would be invalid (to old nodes) are now valid



Soft-forks (1)

Blocks that would be valid are now invalid; thus new blocks created are a subset of the possible blocks that the old rule-set would allow. Both old and new nodes will accept new blocks. However, blocks created by old nodes will be accepted only by old nodes.

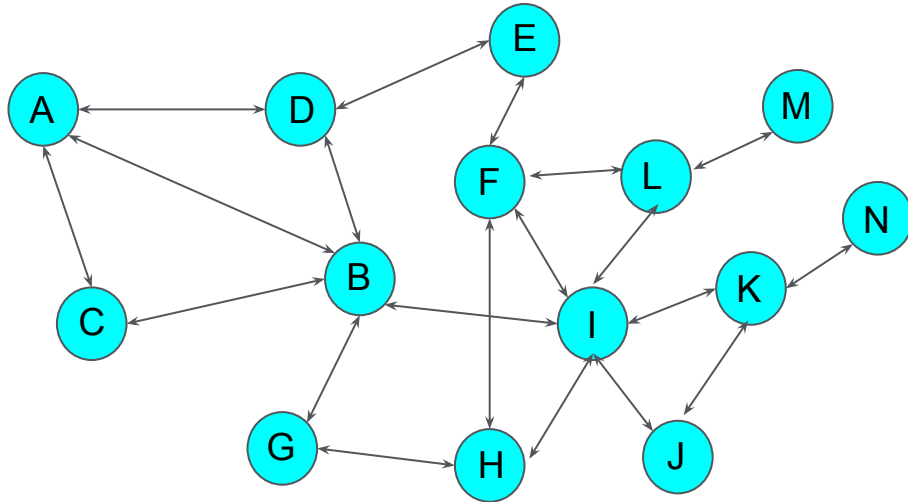
In theory, even 51% of the hashrate would be enough for the new chain since it will consistently (over a period) have the longer chain. Since the longer chain consists of new blocks which are valid by both old and new nodes, the old nodes will switch to the chain consisting new node blocks; thus the blockchain itself remains compatible between all the nodes.

Soft-forks are forward compatible; valid inputs of the new version are also valid by the old version. They do not force old nodes to upgrade or else consider them out of consensus.



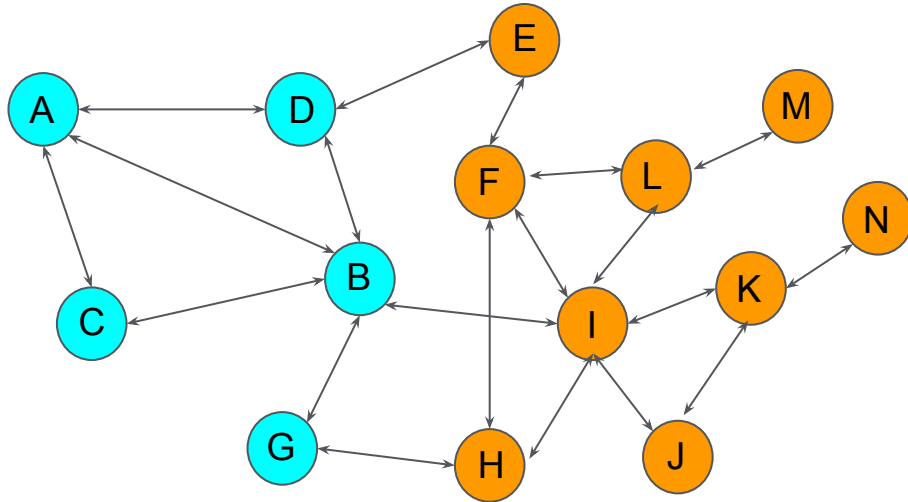
Soft-forks (2)

Initial state of network.

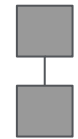


Soft-forks (2)

33% vs 67%



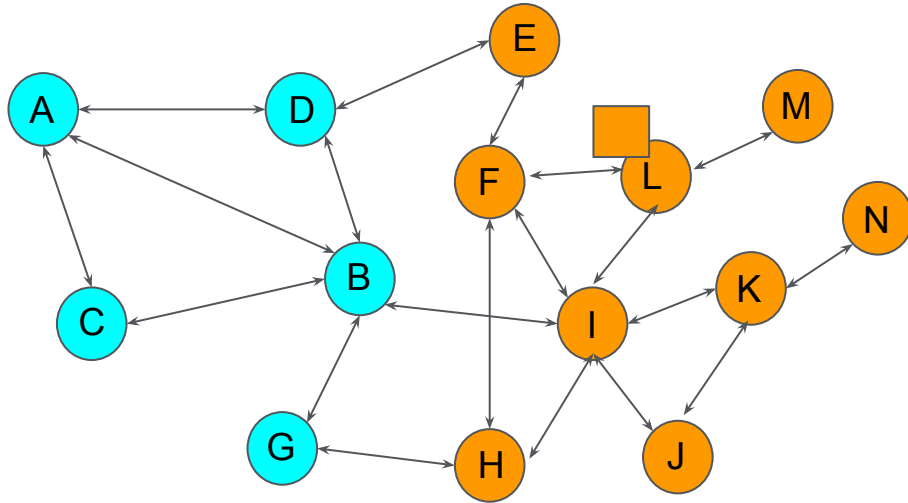
An soft-fork upgrade occurs where 67% of the network uses the new rules.



Soft-forks (2)

33% vs 67%

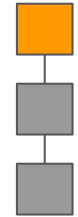
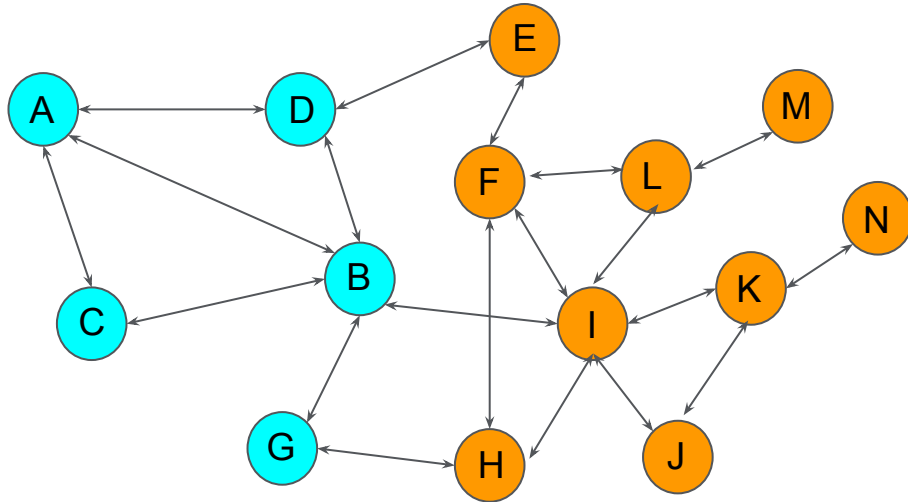
A block is created based on the new rules.



Soft-forks (2)

33% vs 67%

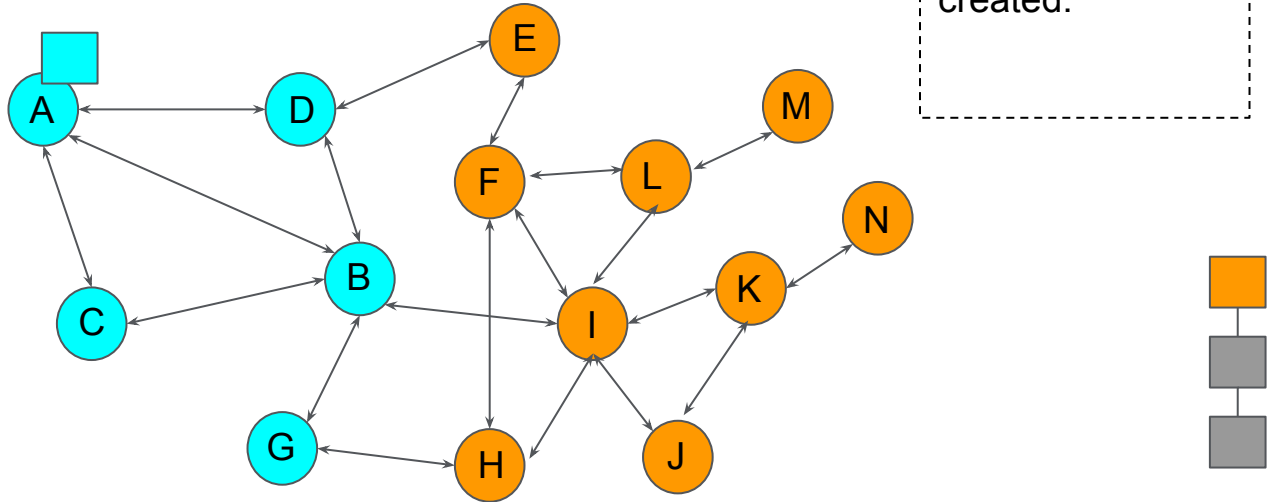
It is accepted by everyone since new rules are a subset of the old rules.



Soft-forks (2)

33% vs 67%

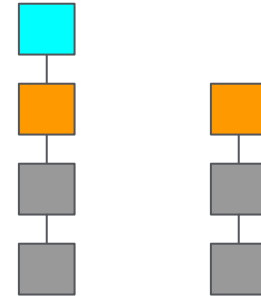
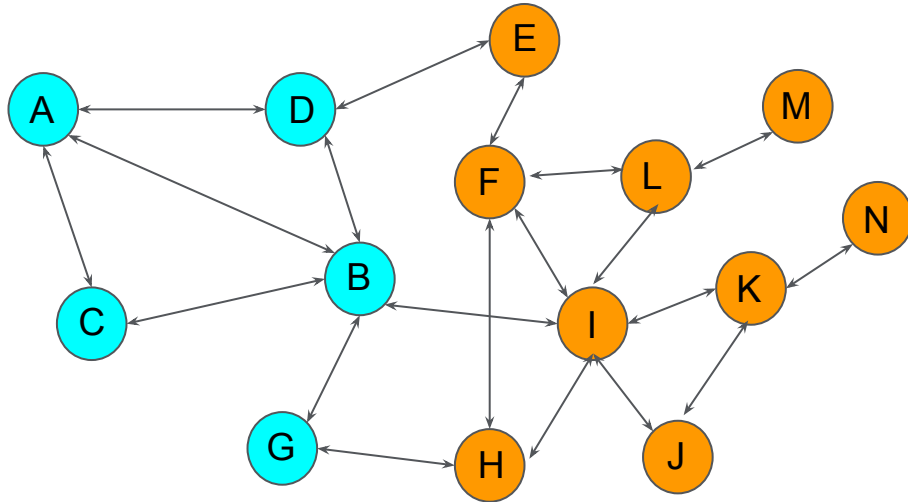
Then a block with the old rules is created.



Soft-forks (2)

33% vs 67%

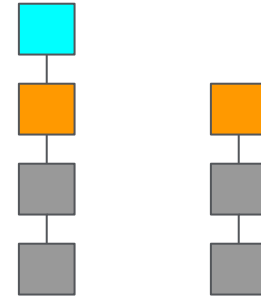
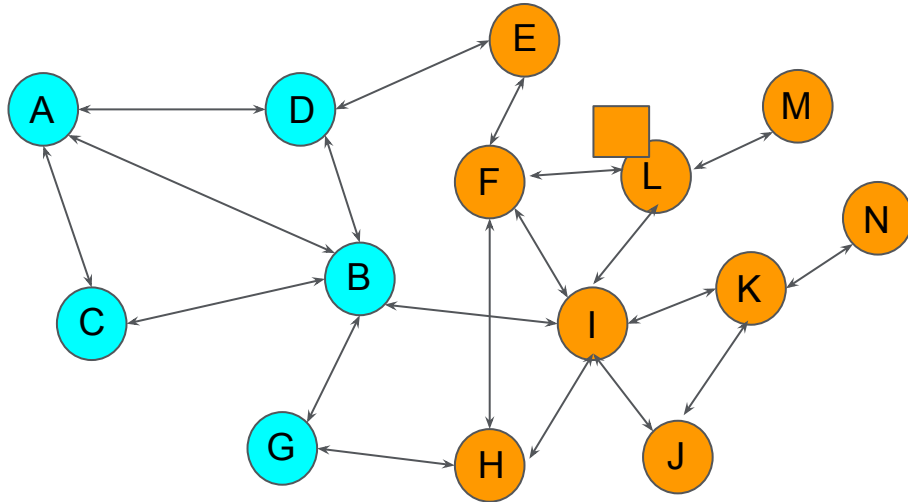
It is only accepted by old nodes.



Soft-forks (2)

33% vs 67%

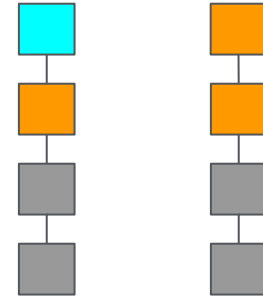
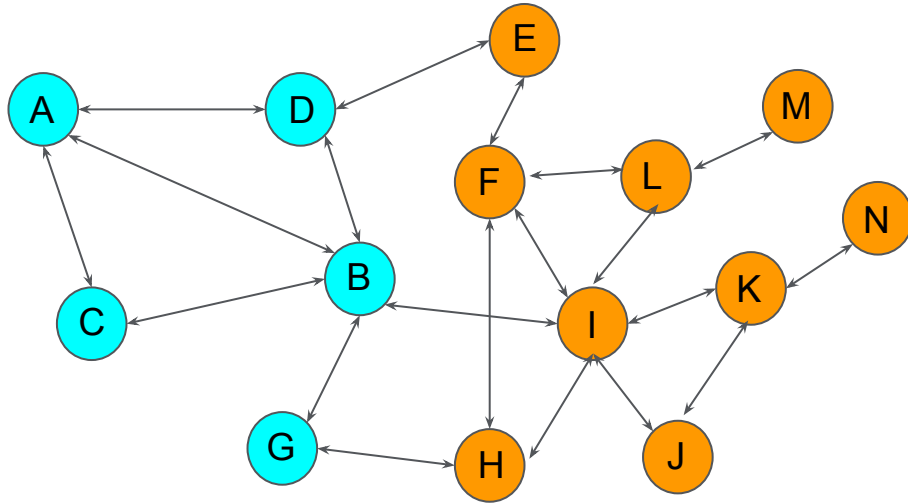
Then another block with the new rules is found.



Soft-forks (2)

33% vs 67%

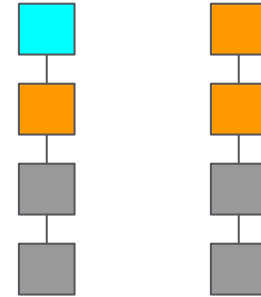
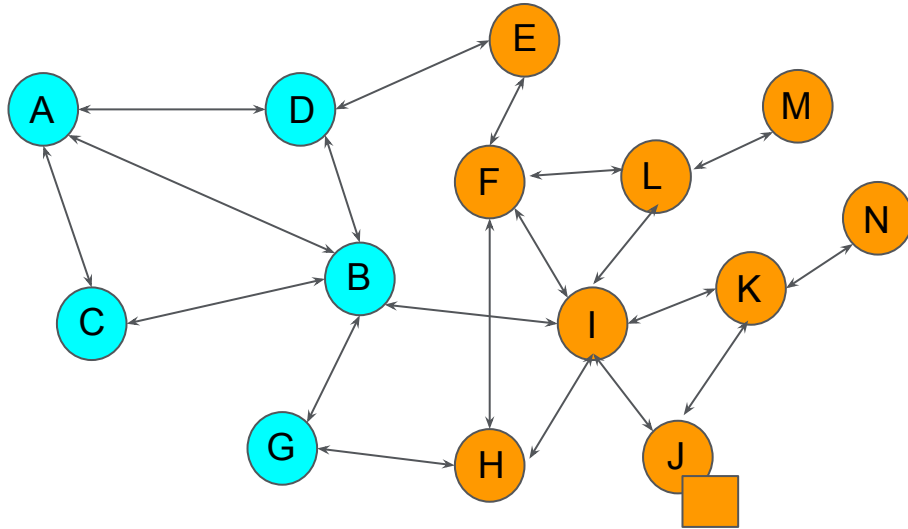
It is accepted only by the new rules since the chains are of equal size.



Soft-forks (2)

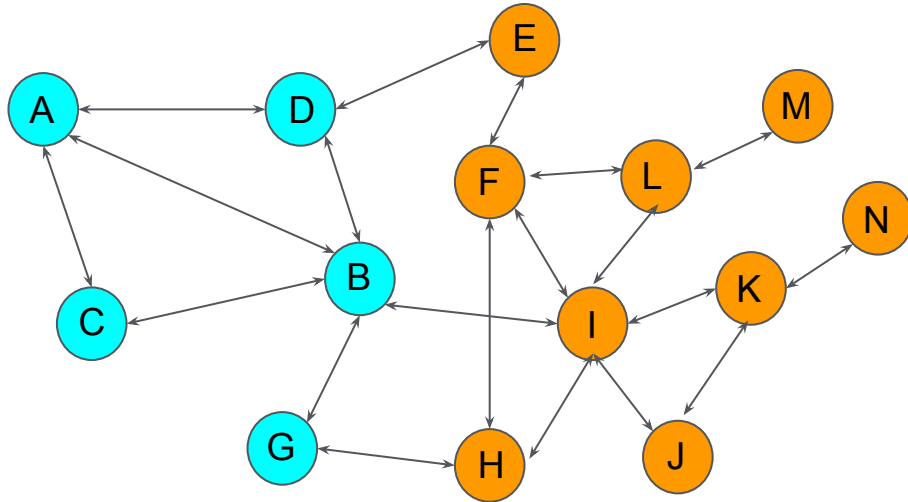
33% vs 67%

Then another block with the new rules is found (67% chance!)



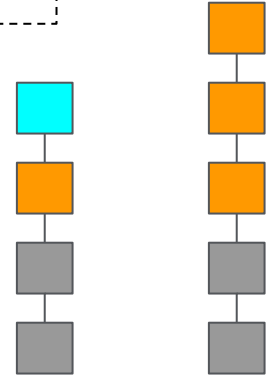
Soft-forks (2)

33% vs 67%



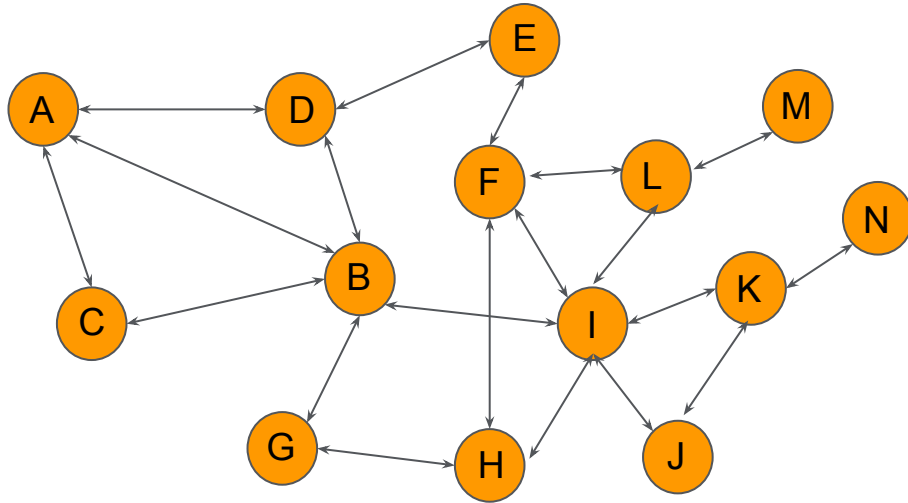
Old nodes are forced to sync with the longest chain (new rules).

Longest valid chain



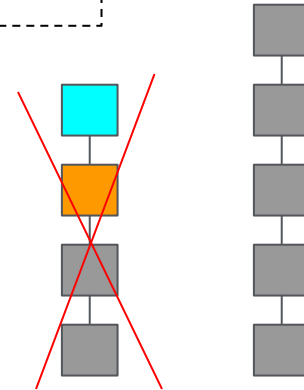
Soft-forks (2)

33% vs 67%



Network is synchronised.

Longest valid chain



Soft-forks (3)

The actual blockchain will always sync to the longest chain and the above mentioned percentages have to do with the hashing rate and thus the miners. However, to other stakeholders like users and merchants a prolonged soft-fork could prove very disruptive.

Specifically, if a merchant is using the old chain for its transactions it is possible that their transactions are ignored when the node switches to the new nodes' (longest) chain. In between, that would lead to *fake* confirmations and potential *double spends*.

Typically, if hashrate is obviously leaning to one side the rest of the network nodes will follow; miners to stop losing rewards and merchants/users to have move consistent transactions.

Note that if the new nodes have 49% or less they will not be able to sustain the longest chain and two incompatible chains will be created that cannot re-sync, leading to a *temporary* hard-fork.



Hard-forks (1)

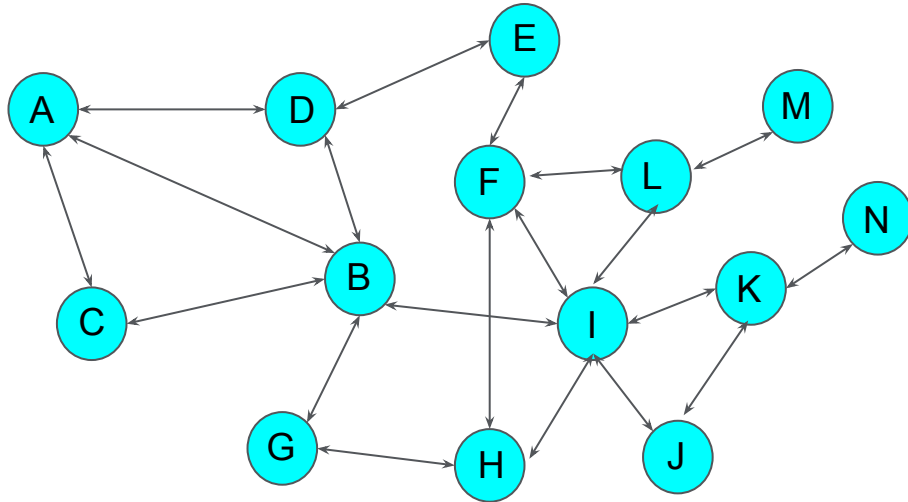
Blocks that would be invalid are now valid; thus new blocks created are a superset of the possible blocks that the old rule-set would allow. Neither old or new nodes will accept blocks created from the others.

Irrespective of the hashing rate this will result in a chain split that will not be able to be resolved unless one of the sides changes software.



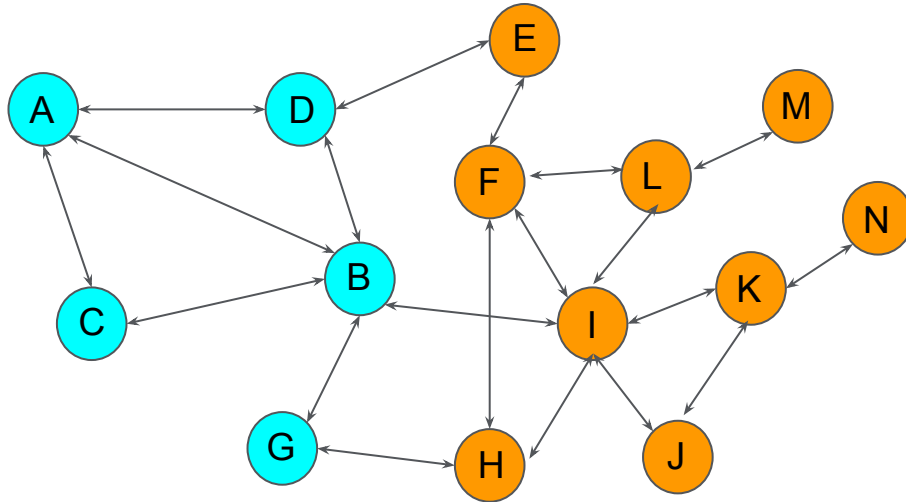
Hard-forks (2)

Initial state of network.

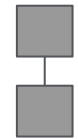


Hard-forks (2)

33% vs 67%



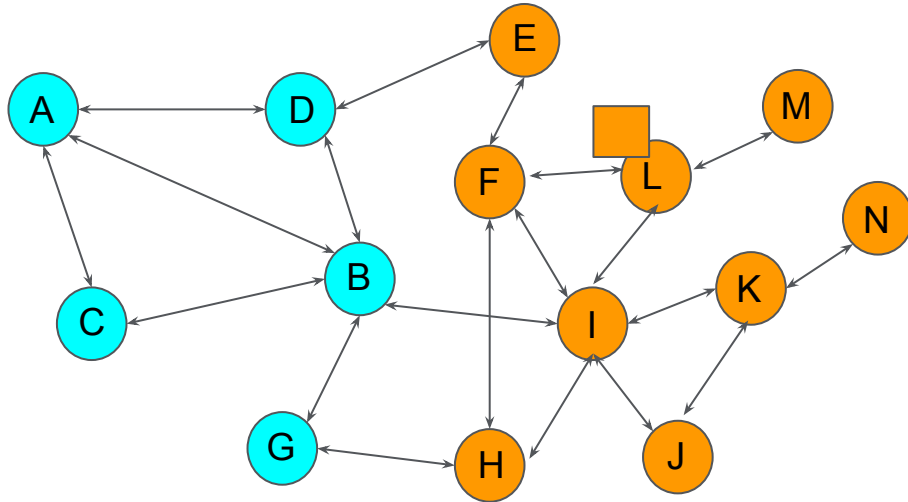
An hard-fork upgrade occurs where 67% of the network uses the new rules.



Hard-forks (2)

33% vs 67%

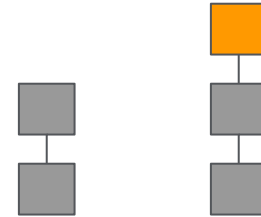
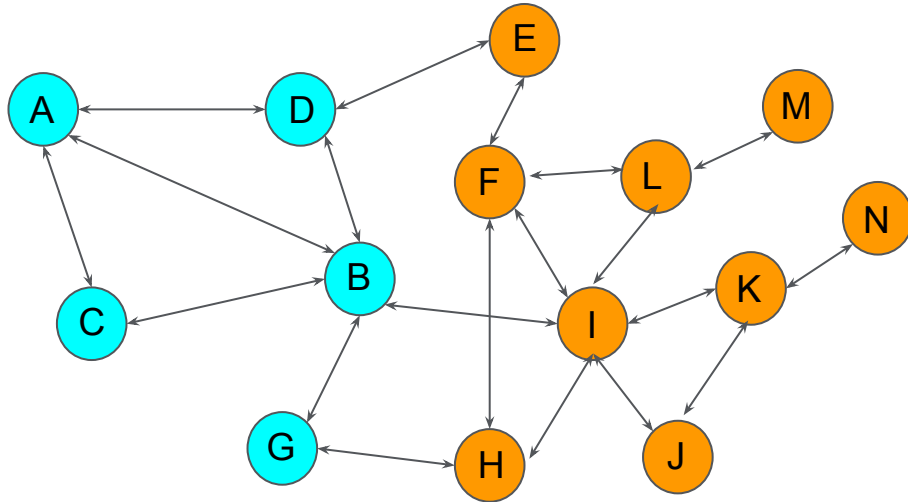
A block is created based on the new rules.



Hard-forks (2)

33% vs 67%

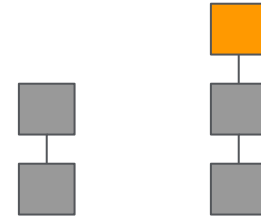
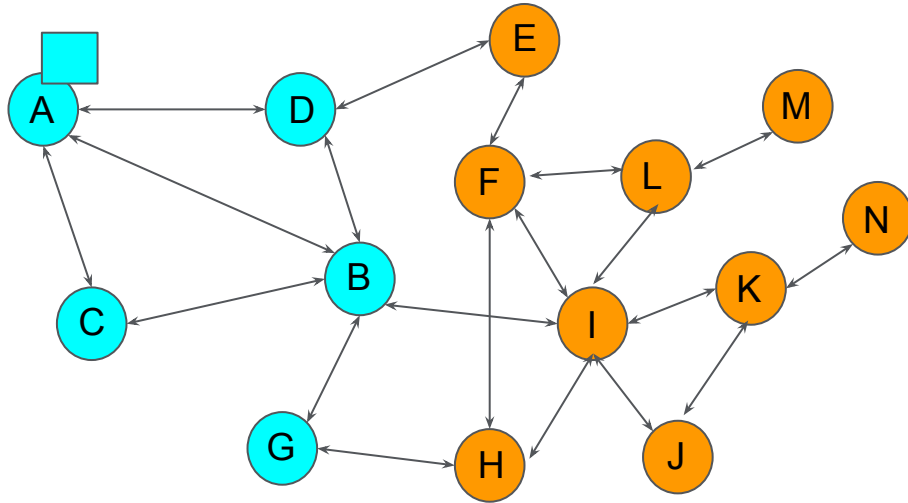
It is accepted only by the nodes with the new rules.



Hard-forks (2)

33% vs 67%

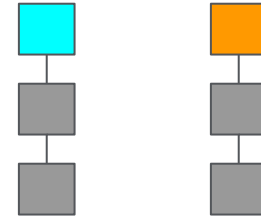
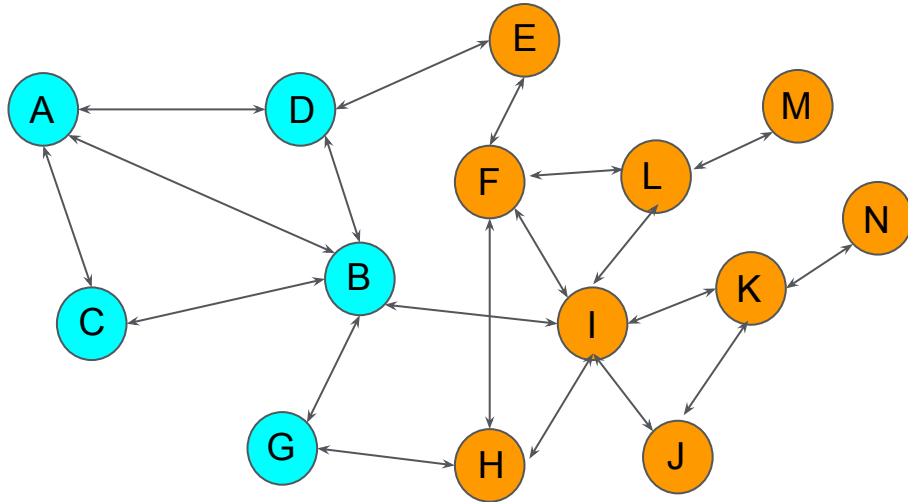
A block is created based on the old rules.



Hard-forks (2)

33% vs 67%

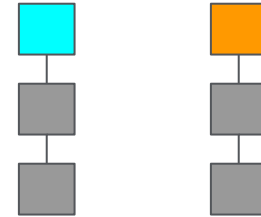
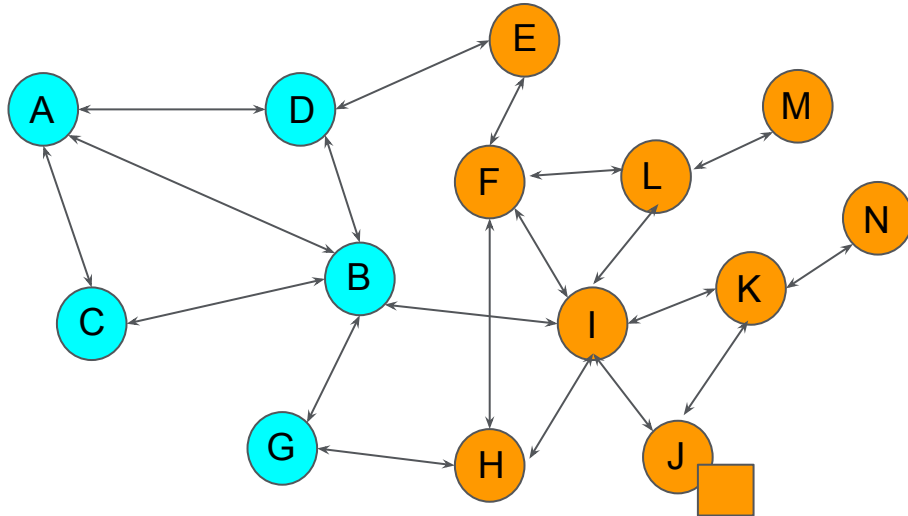
It is accepted only by the nodes with the old rules.



Hard-forks (2)

33% vs 67%

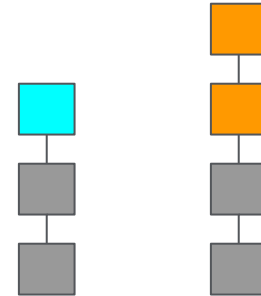
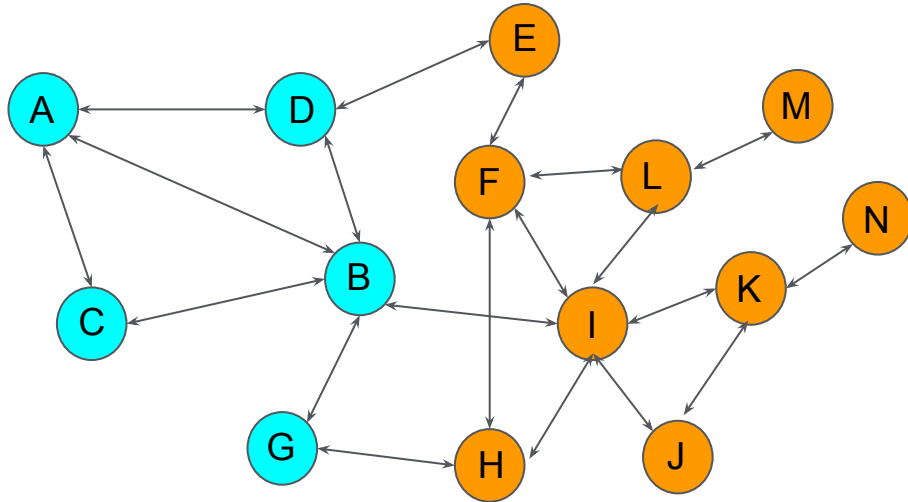
Another block is created based on the new rules.



Hard-forks (2)

33% vs 67%

The old nodes will never accept the incompatible blocks from a longest chain!



Hard-forks (3)

If a hard-fork occurs the network is effectively split in two. The mining hashrate is split in two as well as the merchants and users. If one side does not change their software a hard-fork can permanently split the community in two, effectively having two separate coins from that point onwards.

The same amount of bitcoins will exist in both chains and users will be able to access both. Miners, users and merchants have to choose which side to support and in some cases merchants/users can choose to support both; one of the coins will probably be termed an altcoin and supported as such.

All transactions after the split are in danger of being rolled back (e.g. allow some users to double-spend) if the fork resolves. If not, the trust and thus value of both systems will be diminished.

A chain fork also has potential replay attacks; signed transaction in one chain to be relayed on the other chain. For example, a merchant that gets some bitcoins for a product replays the transaction on the other chain to get the coins of the other chain as well.



Hard-fork examples 1

In June 2010 Bitcoin core v0.2.10 introduced a change to the protocol that was not forward compatible. The *version* messages exchanged by nodes at connection time have changed format and included checksum values.

Since this would lead to a hard-fork ample time was given for all miners, users and merchants to upgrade before the activation of the new feature.

The new feature was activated in February 2012 and it happened without any incident.



Hard-fork examples 2 (1)

In March 2013, Bitcoin core v0.8 switched its UTXOs/indexes database for storing information about blocks and transactions from BerkeleyDB to LevelDB because it was more efficient. However, with the upgrade came an unexpected bug that caused incompatibility between nodes running BerkeleyDB and the new ones running LevelDB.

The bug was that BerkeleyDB had a limit on effectively how many changes it can make to the database while LevelDB did not. The limit was reached and old nodes rejected the block that caused it while new nodes accepted it; blocks that would be invalid where now valid.

The fork was detected quickly by IRC users reporting conflicting block heights on their nodes. The new chain had the majority of the hash rate and thus the old nodes were left behind with a possibility of finding and notifying them being slim.



Hard-fork examples 2 (2)

Major miners were easier to find however and it was quickly agreed that they switch back to v0.7 so that the majority of the hashrate was that of the old nodes. This way thousands of users being on old nodes would not need to upgrade their clients and that would minimize disruption.

Indeed, since it was communicated to most miners that a bug caused a split and, more importantly, since the majority of the hashrate was in the old chain the rest of the miners had strong incentive to revert to the old version as well to be part of the valid chain (and thus get rewards).

There was no political, economic or other incentive to continue with the new chain and thus it died away. Some miners lost their rewards as well as a merchant via a successful double spend but other than that the incident was painless.



Conclusions



Conclusions

- We described the basics of the P2P network initialisation and synchronisation.
- We got an understanding of what blockchain forks are, when they occur and their consequences



Further Reading



Self-assessment exercises

1. Prepare a list with all the forks (soft and hard) that occurred in Bitcoin from the beginning
2. Create a program that given a new block displays the signalling information of that miner. Start by first thinking what are the steps required to accomplish this?



Further Reading

Mastering Bitcoin (Ch. 8 & Ch.10), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch10.asciidoc>

(Bitcoin Mining and Consensus)

Bitcoin Wiki - Soft-forks

<https://en.bitcoin.it/wiki/Softfork>





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming Upgrading and Codebase Konstantinos Karasavvas



Objectives of Session

- Explain how Bitcoin software is upgraded to add new features
- Bitcoins' codebase

In this session we will look into upgrading the network and some history regarding past upgrades. We explain the process and consequences of forking as well as the mechanisms used to upgrade nodes as gracefully as possible.



Agenda

- Upgrading
- Bitcoin's codebase
- Conclusions
- Self-assessment exercises and further reading



Section 1: Upgrading



Upgrading Bitcoin (soft-forks)

During the first years of its existence upgrading Bitcoin involved notifying node owners to upgrade via forums and mailing lists. The community was smaller and more in-line regarding the future of Bitcoin.

As the network grew however coordinating via forums could not scale well so new mechanisms were added to improve the process.

Signalling BIPs to reach consensus before making breaking changes to the rules.

BIP-34: Allow one upgrade at a time (superseded)

BIP-9: Allow several upgrades at a time (superseded)

BIP-8: Similar to BIP-9 but allows automatic activation (current)



BIP-34

The block version was traditionally 1. The BIP suggested that when miners want to support a proposal they would increase the block version to signal that to others and specify in the coinbase input the block height when the upgrade will be activated given it has enough support. The (convention) rules were as follows:

Specify new feature with block version number (say 2) and block height for activation in coinbase

- If 750 out of 1000 blocks have block version of 2 then reject invalid v2 blocks (if no block height included)
 - 510 out of 1000 for testnet3
- If 950 out of 1000 blocks have block version of 2 then reject all block version 1 blocks
 - 750 out of 1000 for testnet3

BIPs activated: BIP-34 (block header v2), BIP-65 (v3), BIP-66 (v4)



BIP-9

BIP-34 allowed only one upgrade at a time and no easy way to reject a proposal to replace it with another. BIP-9 solves these issues with the following (convention) rules:

- The remaining 29 bits of the block version field can be used to signal for 29 proposals
- A structure is defined with:
 - name, usually bipN
 - bit, the block version bit used to signal for this change
 - starttime, time (Median Time-Past, [BIP-113](#)) when signalling can begin
 - timeout, time (MTP) when change is considered rejected if not activated by then
- Threshold for activation is 95%
- Signalling is based on the whole 2016 blocks of a re-target interval
 - if threshold is passed activation occurs one re-target interval later

BIP-9 was used to activate proposal "csv" that contained BIPs 68, 112 and 113 as well as "segwit" with BIPs 141, 143 and 147. There is a [list](#) of all BIP-9 deployments (both past and current ones).



BIP-8

This BIP is very similar to BIP-9. The major differences are:

- Uses block height instead of timestamps for signalling
- Give the option to reject or enforce the upgrade at the end of the timeout
 - lockinontimeout or LOT=false
 - BIP-9 equivalent behaviour
 - LOT=True
 - enforces lock-in

BIP-8 will be used for the activation of the "taproot" proposal.



Blocksize debate

The block size debate have been going on from 2013 but it became more urgent when the Bitcoin's 1MB block size started to fill up completely. As a consequence many transactions could not make it in the current block and had to wait for the next. Miners, give priority to transactions that give more fees and thus fees increased considerably (up to \$2-3 on June 2017) in order to get your transaction in the current block.

Two major camps suggested different solutions*:

- Increase the block size from 1MB to 2MBs (or more)
 - straight-forward software change **but** requires a hard-fork!
- Activate “segwit” a set of proposals that includes, among others, an effective increase of the transactions that fit in a block by x1.8
 - more involved software change that changes the transaction structure **but** achieves it with a soft-fork

* Several suggestions were made, attempted and failed. We only mention the major ones.

User-Activated Soft-Fork (UASF)

The “segwit” proposal that used BIP-9 for signalling and consisting of BIPs 141, 143, 144 and 145 is contentious and it seems that it will not be activated until 15-Nov-2017 which is its expiration date.

According to some statistics more than 80% of the nodes are signalling for “segwit” and thus the rationale of the UASF is that node owners want it but miners block it.

Thus, some supporters of “segwit” created [BIP-148](#) (and the code that implements it!) as an alternative mechanism for signalling “segwit”. All it does, is to not accept/propagate blocks that do not signal for segwit. That will make activation easier and more importantly user-activated rather than miner-activated as was usually the case. With users it refers to everyone **that runs a Bitcoin node**; effectively, allowing every stakeholder to *vote* and not only the miners (but also merchants, exchanges, miners or just users).

The activation threshold is 80% and it will activate on 1-Aug-2017 (MTP) if >80% runs the UASF code and signals for “segwit”. Note that since anyone can run a Bitcoin node UASF is vulnerable to sybil attacks.

Segwit2x

An agreement made end of May 2017 in New York (NYA) between major Bitcoin stakeholders (exchanges, miners, merchants, etc.) tried to compromise between big blocks and segwit. The NYA would have 2 phases. First nodes would signal for 'segwit' and if >80% majority segwit would lock-in and be activated. Then after 3 months the second phase would take place with a hard-fork that increases the block size to 2MB.

The proposal had strong support (with 58 signatories) but later on many signatories backed off their commitment for the 2MB increase since they found that more testing is required for a hard-fork.

That increased support for segwit and another proposal was suggested, called *segsignal*.



Segsignal or MASF

BIP-91 proposal was compatible with the NYA and was created to decrease required consensus from 95+% to 80+%. Support was overwhelming with more than 90% and thus the upgrade was locked.

Then another group, led by ViaBTC mining pool, used the UASF's date of activation (1st Aug) to hard-fork Bitcoin, increasing the block size to 8MBs, creating Bitcoin Cash.

Segwit activated on block 481,824 (24 Aug 2017).

In November, the second phase was expected but many signatories backed off their commitment for the 2MB increase and it never happened; that led to more contention with more people supporting Bitcoin Cash.



Taproot upgrade in progress

- name: taproot
- bit: 2
- startheight: 693504 (~2021 July 23rd)
 - starttime
- timeoutheight: 745920 (~2022 July 22nd / 1 year after signalling begins)
 - timeout
- threshold: 1815 out of 2016 blocks (90%)
- lockinontimeout: no consensus as of 8 March 21

Source: https://en.bitcoin.it/wiki/Taproot_activation_proposal_202102



Section 2: Bitcoin's Codebase



Bitcoin codebase (1)

The architecture and abstractions discussed were developed from people trying to design new blockchain systems. People that understood how Bitcoin works tried to componentize it in an effort to design cleaner systems.

In general, it is good practice to go through the codebase of a project in order to get a deep understanding of how it works. This is quite challenging for a relatively large project like Bitcoin. More importantly, the codebase (although improved considerably) still misses encapsulations and abstractions with a lot of components being tightly coupled together.

There have been attempts to improve this, like abstracting the consensus rules in a separate library ([libconsensus](#)) but many of these were abandoned. Notably, in v0.17 the process running the GUI is being separated from the process running the node; and it is planned to have the wallet code in yet another process in the future.

Lack of proper abstractions make the project much more difficult to test and as a result people do as little changes as possible in fear of introducing new bugs to the codebase. As a result Bitcoin's code base carries a lot of technical debt with it and it is even more challenging for a new developer to get comfortable and develop on it.



Bitcoin codebase (2)

V0.13.1 -- C/C++: 573 files with 175091 lines of code plus 17581 lines of comments

```
kostas@kostas-pc:~/crypto/bitcoin$ cloc src/
1292 text files.
1143 unique files.
682 files ignored.

http://cloc.sourceforge.net v 1.60 T=3.71 s (172.7 files/s, 87842.8 lines/s)
-----
Language          files      blank      comment      code
-----
C/C++ Header      288         6106         8866         98022
C++                285        12831         8715         77069
Bourne Shell       14         6474         7012         38516
Bourne Again Shell 4           3482         5198         18995
m4                 11         1179          218         10901
make              10          1162         2904         8871
C                  16           607          512         6082
HTML               3            85            0          1136
Objective C++      2            39            18           168
YAML               2            10            0           105
CSS                1            10            1            78
Python             3            11            8            53
Java               1             7            19            34
-----
SUM:                640        32003        33471        260030
-----
```



Bitcoin codebase (3)

V0.1.5 -- C/C++: 26 files with 14775 lines of code plus 1695 lines of comments

```
kostas@kostas-pc:~/crypto/oldbitcoin$ cloc bitcoin/
 33 text files.
 33 unique files.
 44 files ignored.

http://cloc.sourceforge.net v 1.60 T=0.12 s (218.7 files/s, 163585.8 lines/s)
-----
Language          files          blank          comment          code
-----
C++                10             2199             1131             8833
C/C++ Header      16             1441              564             5942
make               1               26                5                52
-----
SUM:               27             3666             1700            14827
-----
```

Bitcoin codebase (4)

If you want to study Bitcoin's codebase, e.g. in order to be able to contribute, you will need to study the codebase. Your success would depend on your C/C++ experience including debugging tools and your perseverance.

To get an idea of the pace of contributions, the code changes from v0.16 to v0.17 involved*:

- 1225 non-merge commits (6.3/day)
- 135 unique commit authors (67 new authors)
- 958 files changed, +45370/-65542 (**568/day**)

You may want to start with an older and simpler version of the software albeit with a smaller feature set. There is a relatively [detailed explanation of v0.3.23](#) in the bitcointalk forum so you could use this version to get a glimpse of the implementation. Of course since the codebase increased almost tenfold it would just be the starting point of such a journey.

* Statistics from <http://diyhpl.us/wiki/transcripts/london-bitcoin-devs/jnewbery-bitcoin-core-v0.17/>



Tinychain

“Tinychain is a pocket-sized implementation of Bitcoin. Its goal is to be a compact, understandable, working incarnation of the Nakamoto consensus algorithm at the expense of advanced functionality, speed, and any real usefulness.”

Tinychain condenses the very basic in ~800 lines of python code.



Conclusions



Conclusions

- We explained the different methods and mechanisms used to upgrade the Bitcoin software / network



Further Reading



Self-assessment exercises

1. Provide a (detailed) walkthrough of the proposals created to resolve the block size debate
2. What would the core developers need to do to fix the BerkeleyDB to LevelDB bug?
3. Split Bitcoin according to the architectural components that we mentioned. Which part of Bitcoin fits in each component?



Further Reading

Mastering Bitcoin (Ch. 8 & Ch.10), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc>

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch10.asciidoc>

(Bitcoin Mining and Consensus)

Signalling BIPs

Old: <https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki>

New: <https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki>

Scorex

<https://github.com/input-output-hk/Scorex>



UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming Advanced Topics, Part 1

Konstantinos Karasavvas



Objectives of Session

- Introduce more advanced concepts like HTLC contracts and Atomic swaps
- Describe several state of the art concepts

In this session we introduce some of the more advanced Bitcoin concepts with script examples so that people can work on their own implementations.



Agenda

- Native Wallet Descriptors
- Partially Signed Bitcoin Transactions (PSBTs)
- Hashed Time-Locked Contracts (HTLCs)
- Atomic Swaps
- Conclusions
- Self-assessment exercises and further reading



Section 1: Native Descriptor Wallets



Output Script Descriptors

The Bitcoin Core wallet primarily stores private keys. It uses these keys to create public keys and addresses. This works fine for single key scripts like those needed for P2PK, P2PKH and P2WPKH. However, Bitcoin Script supports arbitrary scripts (P2SH and P2WSH) which cannot really be expressed in the wallet right now. Single key descriptions cannot even express extended keys properly, e.g. using derivation paths.

Consider a multisignature script created from another wallet. You can import that address to your wallet and see the funds but you will not be able to spend even if you have imported the keys since there is no way to 'describe' how these keys were combined to create that address.

Output script descriptors are strings that contain all the necessary information to spend an output script. Some examples (from linked resource) are:

```
pkh(02c6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5)
```

```
combo(0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
```

```
sh(multi(2,022f01e5e15cca351daff3843fb70f3c2f0a1bdd05e5af888a67784ef3e10a2a01,03acd484e2f0c7f65309ad178a9f559abde09796974c57e714c35f110dfc27ccbe))
```

```
pkh([d34db33f/44'/0'/0']xpub6ERApfZwUNrhLCkDtChtCxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRkrGZw4koxb5JaHWkY4ALHY2grBGRjaDMzQLcgJvLJuZZvRcEL/1/*)
```



Native Descriptor Wallets

Output script descriptors were introduced in v0.17 of Bitcoin Core and several RPC commands, like `listunspent` and `getaddressinfo` where updated accordingly.

Descriptor Wallets are introduced in v0.21 and explicitly store output script descriptors in the wallet*. These descriptors are then used to generate the addresses.



Section 2: Partially Signed Bitcoin Transactions (PSBTs)



Partially Signed Bitcoin Transactions

PSBTs described in [BIP-174](#) specifies a binary format that allows wallets to exchange unfinished transactions in a standardised way. A PSBT describes all the UTXOs that need to be spend and the outputs that will receive the funds. All necessary information to spend a UTXO can be included as well.

An example where PSBTs are very useful is when we need to sign a multisignature script. The partial transaction can be shared to individual signers and then the results could be merged to produce the final transaction.



Section 3: Hashed Time-Locked Contracts



Hashlocks

A hashlock is a type of locking script that restricts the spending of an output until a specific piece of data (aka as pre-image or passphrase) is publicly revealed. The passphrase can be shared by any means. All hashlock outputs using the same passphrase can then be spent.

Example locking script:

```
OP_HASH256 <passphrase_hash> OP_EQUAL
```

This makes it possible to create multiple outputs locked with the same hashlock and when one is spent the rest will also be available for spending (since by spending one the passphrase will be revealed). Effectively, by spending one such output you *share* the passphrase.

Of course, since the passphrase will become public, everyone will be able to spend the rest of the outputs, which is not very useful. Thus, outputs protected by hashlocks are typically also protected by specific signatures so that only the owners of the corresponding keys could spend the remaining outputs. This is similar to what 2FA offers (something one owns and something one knows).

Example:

```
OP_HASH256 <passphrase_hash> OP_EQUALVERIFY OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY  
OP_CHECKSIG
```



HTLC

A Hashed Time-Locked Contract ([BIP-199](#)) is a combination of a hashlock and a timelock that requires the receiver of a payment to either provide a passphrase or forfeit the payment allowing the sender to take the funds back.

Example:

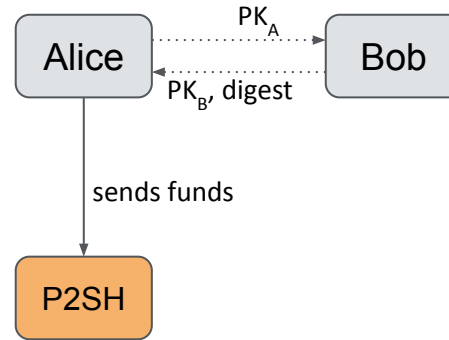
```
OP_IF
  OP_SHA256 <passphrase_hash> OP_EQUALVERIFY OP_DUP OP_HASH160 <receiver PKH>
OP_ELSE
  100 OP_CHECKSEQUENCEVERIFY OP_DROP OP_DUP OP_HASH160 <sender PKH>
OP_ENDIF
OP_EQUALVERIFY
OP_CHECKSIG
```

The above locking script would be created by both sender and receiver collaborating. The receiver knows the passphrase, also called pre-image, but only shares its hash also called digest. The sender can then send some funds to that P2SH address. The receiver can claim the funds if he reveals (in the blockchain) the passphrase. If not, after 100 blocks pass the sender can claim the funds.



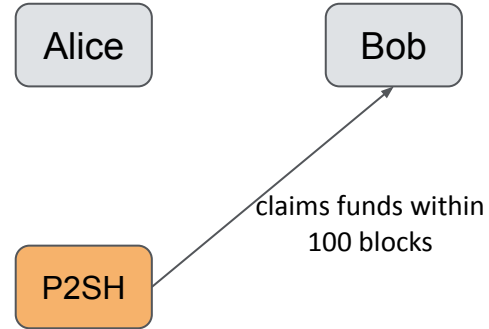
HTLC Example

- Assumptions
 - Alice (sender) and Bob (receiver) exchange public keys
 - Alice and Bob agree upon a timeout threshold
 - Bob sends the passphrase_hash (digest) to Alice
 - They can both create the script and P2SH address
 - Alice sends funds to the new P2SH address

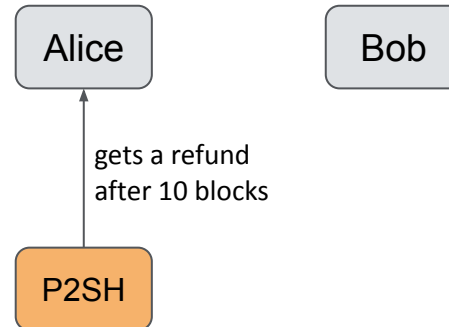


HTLC Example

- Scenario 1
 - Bob claims the funds and in doing so reveals the passphrase



- Scenario 2
 - Bob does not claim the funds until the agreed timeout
 - Alice takes the funds back



HTLC Applications

HTLC transactions are a safe and cheap method of exchanging secrets for money over the blockchain. Applications include Atomic Swaps, Lightning Network, [Zero-knowledge contingent payments](#) and potentially several others.



Section 4: Atomic Swaps



Atomic Swaps

Atomic Swaps is a way of trustlessly exchanging funds between different blockchains. You can swap funds in a predetermined exchange rate. For example Alice wants to send 1 BTC to Bob in the Bitcoin blockchain and receive 100 LTC from Bob in the Litecoin blockchain. It is important that these two transactions happen atomically, either both happen or none.

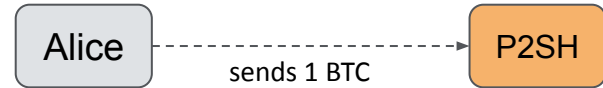
To accomplish that we can use two HTLC contracts, one in each blockchain. The same passphrase should be used, thus once the funds from one of the blockchains is claimed (passphrase revealed) it can immediately be claimed in the other.

- Assumptions
 - Alice and Bob exchange public keys on both Bitcoin and Litecoin
 - Alice and Bob agree upon a timeout threshold, say 48 hours
 - Alice knows of a passphrase (pre-image) which is hashed to produce a *digest*



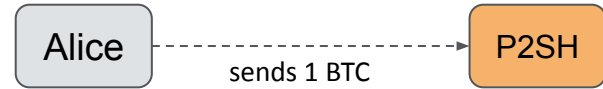
Atomic Swap Example (1)

- Bitcoin network
 - Alice creates a transaction moving coins to an Output₁ that can be redeemed by:
 - revealing the passphrase and Bob's signature
 - both Alice's and Bob's signatures
 - Does not broadcast!



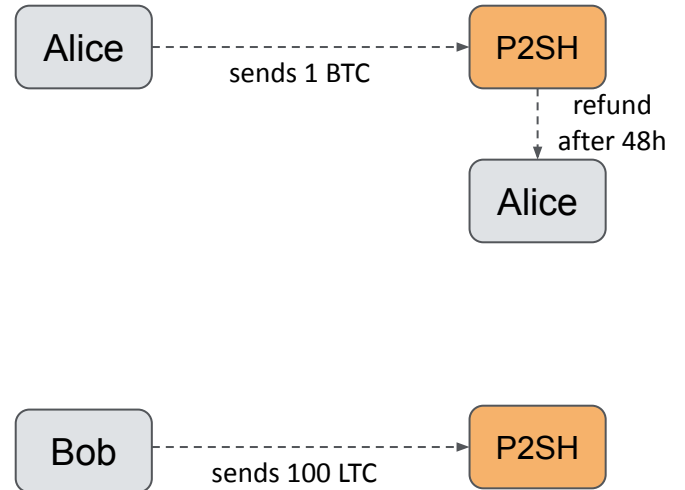
Atomic Swap Example (1)

- Bitcoin network
 - Alice creates a transaction moving coins to an Output₁ that can be redeemed by:
 - revealing the passphrase and Bob's signature
 - both Alice's and Bob's signatures
 - Does not broadcast!
- Litecoin network
 - Bob creates a transaction moving coins to an Output₂ that can be redeemed by:
 - revealing the passphrase and Alice's signature
 - both Alice's and Bob's signatures
 - Does not broadcast!



Atomic Swap Example (2)

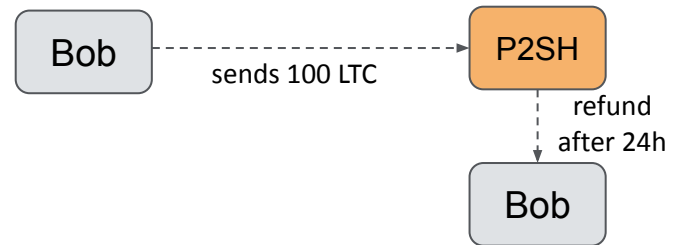
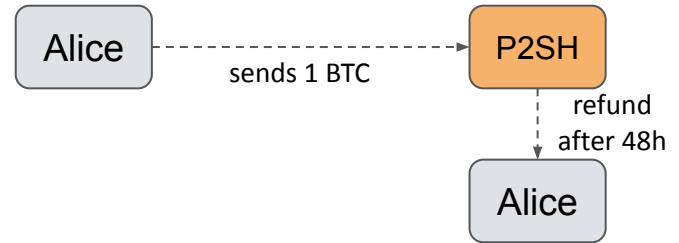
- Bitcoin network
 - Alice creates a second transaction returning the coins from Output₁ to Alice (an address that she owns)
 - timelocked for 48 hours
 - Passes this transaction to Bob to sign
 - Bob signs the refund transaction and returns it to Alice



Atomic Swap Example (2)

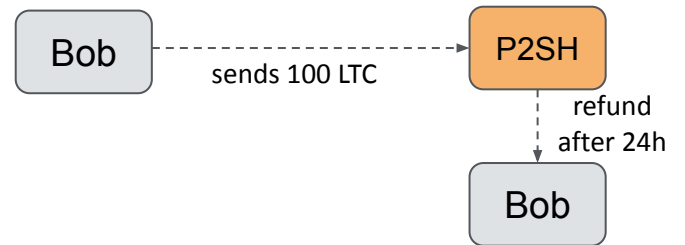
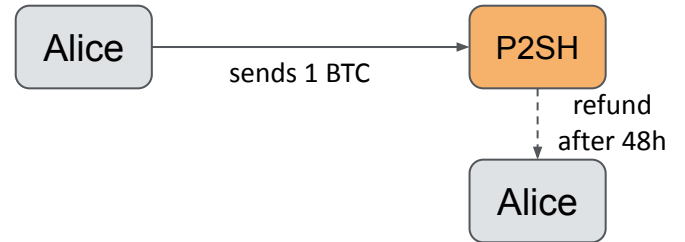
- Bitcoin network
 - Alice creates a second transaction returning the coins from Output₁ to Alice (an address that she owns)
 - timelocked for 48 hours
 - Passes this transaction to Bob to sign
 - Bob signs the refund transaction and returns it to Alice

- Litecoin network
 - Bob creates a second transaction returning the coins from Output₂ to Bob (an address that he owns)
 - timelocked for 24 hours
 - Passes this transaction to Alice to sign
 - Alice signs the refund transaction and returns it to Bob



Atomic Swap Example (3)

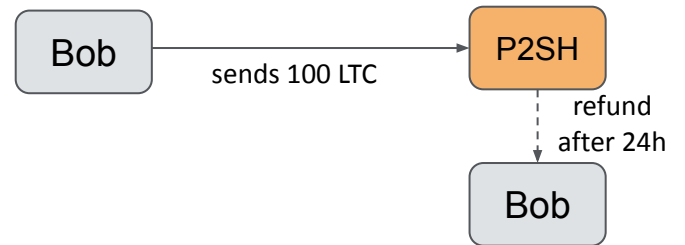
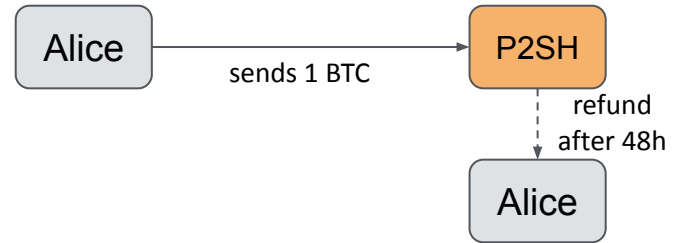
- Bitcoin network
 - Alice broadcasts the first transaction (sends 1 BTC to Output₁)



Atomic Swap Example (3)

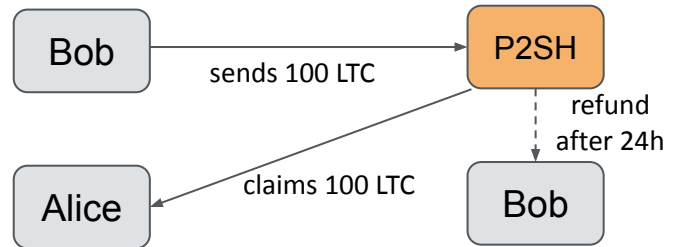
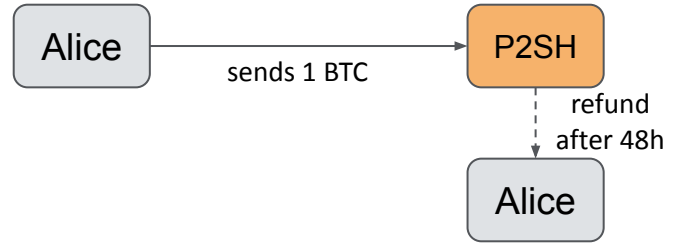
- Bitcoin network
 - Alice broadcasts the first transaction (sends 1 BTC to Output₁)

- Litecoin network
 - Bob broadcasts the first transaction (sends 100 LTC to Output₂)



Atomic Swap Example (4)

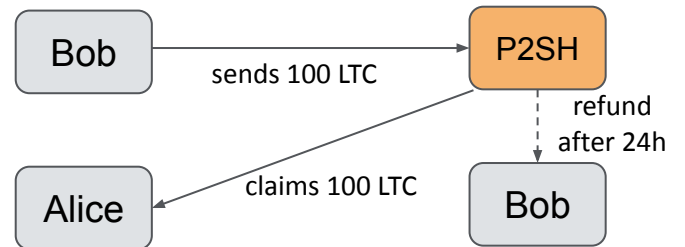
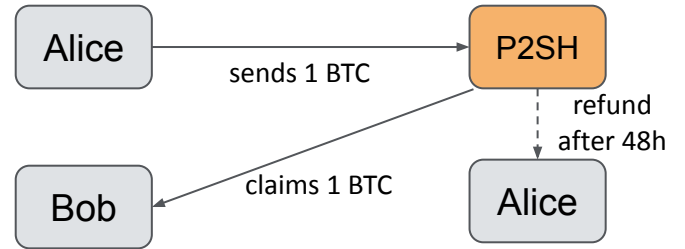
- Litecoin network
 - Alice reveals the passphrase, signs and claims the 100 LTC (Output₂)



Atomic Swap Example (4)

- Bitcoin network
 - Bob uses the passphrase, signs and claims the 1 BTC (Output₁)

- Litecoin network
 - Alice reveals the passphrase, signs and claims the 100 LTC (Output₂)



Atomic Swaps

Note that the above ordering is not strict in any sense. As long as the refund transactions are both signed before the passphrase is revealed everyone is safe.

If Bob does not send the LTC, Alice will be able to get her 1 BTC back after 48 hours by broadcasting her refund transaction.

If Alice does not send the BTC, Bob will be able to get his 100 LTC back after 24 hours by broadcasting his refund transaction.

It is important to understand that active participation is required in this exchange. For example, if Bob does not use the passphrase to claim the bitcoin in time after the passphrase is revealed, Alice can use her refund transaction and also get her bitcoin back!

Atomic swaps allow for trustless exchange between assets of different blockchains. That allows for trustless decentralized exchanges. But how will users find each other orders?



Conclusions



Conclusions

- We introduced HTLC contracts and its components
- We explained what are atomic swaps as well as how they can be implemented



Further Reading



Self-assessment exercises

- Implement the example HTLC scenario of slides 11-13. Create the appropriate scripts for both Alice and Bob.
- Write the HTLC locking script that Alice needs to create for the atomic swap step of slide 18.
- Write the unlocking script that Alice needs to use to claim the litecoin as in slide 24.
- Try to design a platform that would facilitate atomic swaps between Bitcoin and Litecoin. Think about it holistically and in practical terms; i.e. how would you design and implement such a platform. Keep a note of all the potential difficulties that might come up and try to find solutions.



Further Reading

Output Descriptors

<https://github.com/bitcoin/bitcoin/blob/master/doc/descriptors.md>

<https://achow101.com/2020/10/0.21-wallets>

PSBTs

<https://bitcoinops.org/en/topics/psbt/>

<https://github.com/bitcoin/bips/blob/master/bip-0174.mediawiki>

Atomic Swaps

https://en.bitcoin.it/wiki/Atomic_swap

<https://medium.com/blockchainio/what-are-atomic-swaps-bc1d034634c9>

<https://blockgeeks.com/guides/atomic-swaps/>





UNIVERSITY *of*
NICOSIA

BLOC-521 Digital Currency Programming Advanced Topics, Part 2

Konstantinos Karasavvas



Objectives of Session

- Explain how payment channels are leveraged in the Lightning Network
- Describe several state of the art concepts

In this session we introduce some of the more advanced Bitcoin concepts with script examples so that people can work on their own implementations.



Agenda

- Payment Channels
- Lightning Network
- Conclusions
- Self-assessment exercises and further reading



Section 1: Payment Channels

Payment Channels

Payment channels is a class of techniques that allows two participants to make multiple Bitcoin transactions without committing all of those transactions to the blockchain; i.e. most of the transactions are off-chain.

The Bitcoin wiki [lists](#) several approaches to implement Payment Channels. We will only go through some of them including the one currently used in the Lightning Network.

- Nakamoto high-frequency transactions
- **Spillman-style payment channels**
- **CLTV-style payment channels**
- **Poon-Dryja payment channels**
- Decker-Wattenhofer duplex payment channels
- Decker-Russell-Osuntokun eltoo Channels
 - eltoo; requires SIGHASH_NOINPUT

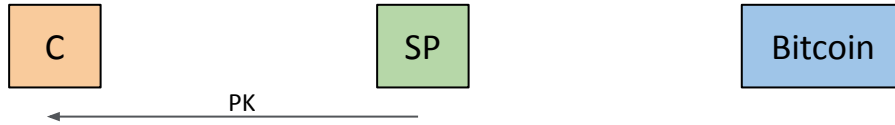


Spilman-style payment channels

Assume that a customer wants to watch a video from a streaming service and pay by the minute. The frequency of the payment (as well as the tx fees) makes it impossible to work with on-chain Bitcoin transactions.

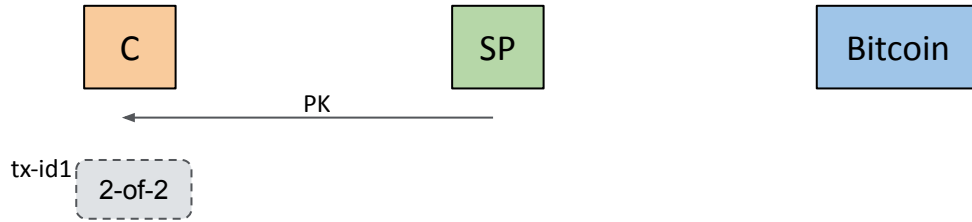


Spilman-style payment channels



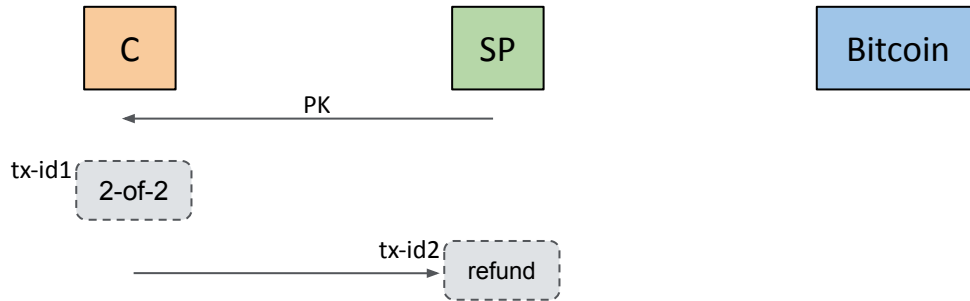
- (1) Service Provider (SP) gives their public key to Customer (C)

Spilman-style payment channels



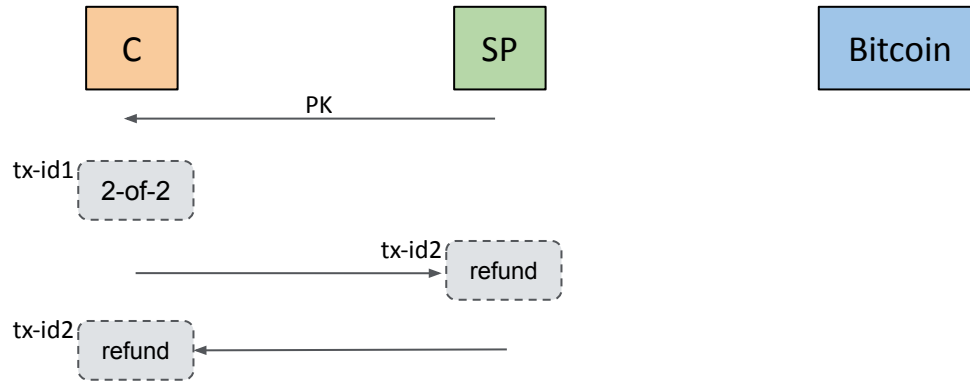
- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet

Spilman-style payment channels



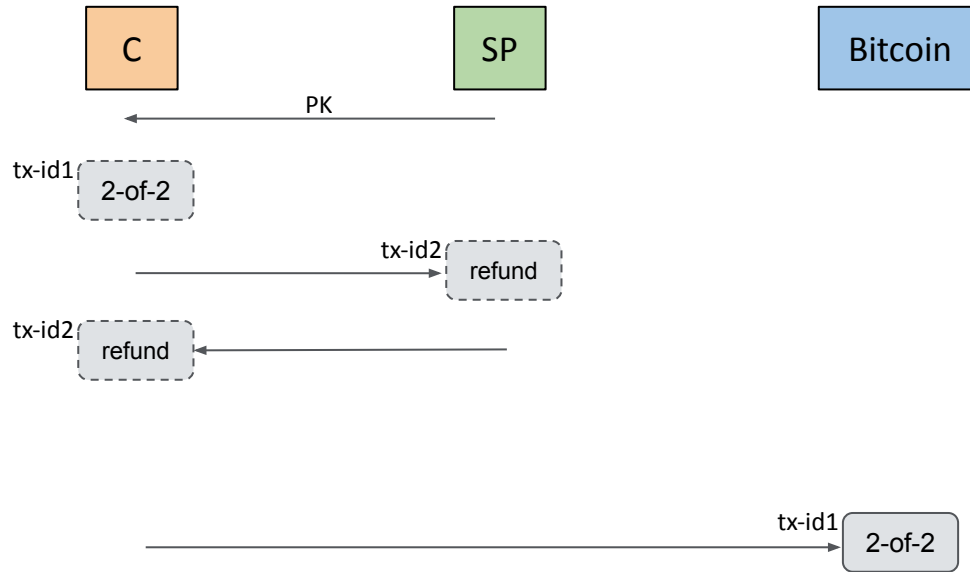
- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet
- (3) C creates a refund tx-id2 that sends the funds back to C `nLockTime`'ed 12 blocks in the future (~2 hours) and sends to SP to sign

Spilman-style payment channels



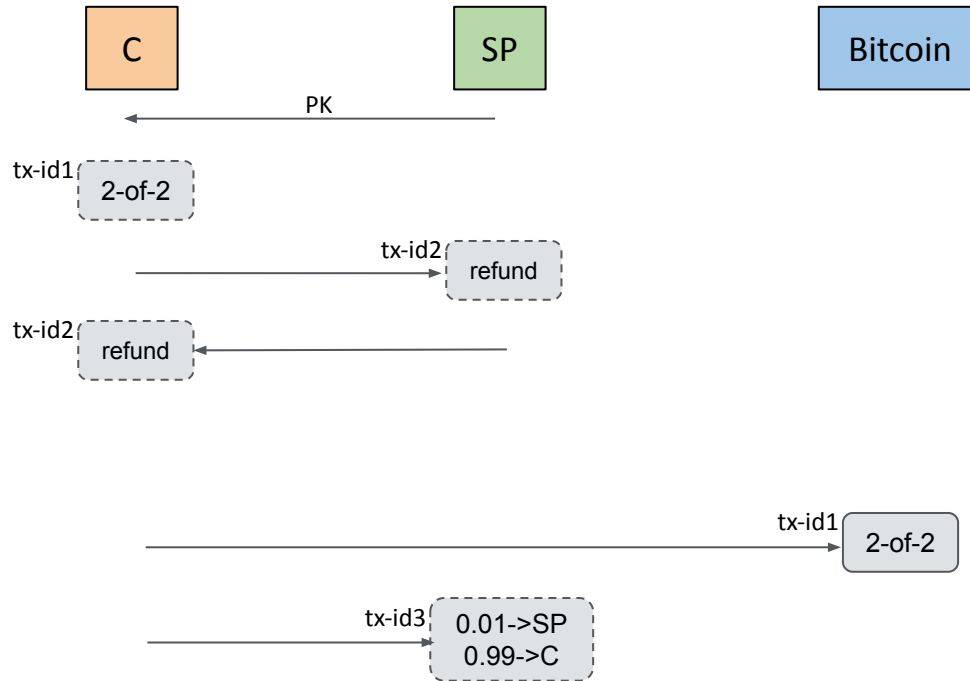
- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet
- (3) C creates a refund tx-id2 that sends the funds back to C `nLockTime`'ed 12 blocks in the future (~2 hours) and sends to SP to sign
- (4) SP signs refund tx and returns

Spilman-style payment channels



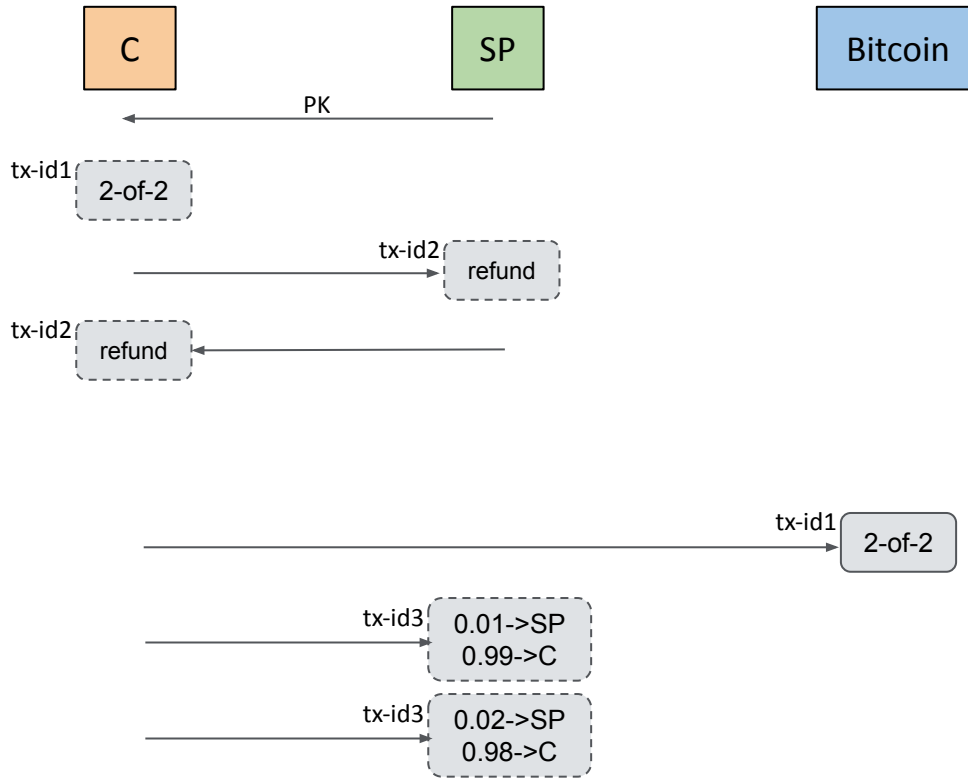
- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet
- (3) C creates a refund tx-id2 that sends the funds back to C `nLockTime`'ed 12 blocks in the future (~2 hours) and sends to SP to sign
- (4) SP signs refund tx and returns
- (5) Then C broadcasts tx-id1 that sends the funds to the 2-of-2 address

Spilman-style payment channels



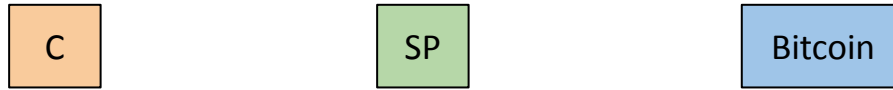
- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet
- (3) C creates a refund tx-id2 that sends the funds back to C `nLockTime`'ed 12 blocks in the future (~2 hours) and sends to SP to sign
- (4) SP signs refund tx and returns
- (5) Then C broadcasts tx-id1 that sends the funds to the 2-of-2 address
- (6) C creates a tx-id3 which spends the 2-of-2 to send 0.01 to SP and 0.99 to C, signs and sends to SP (paying for 1 minute of service)

Spilman-style payment channels

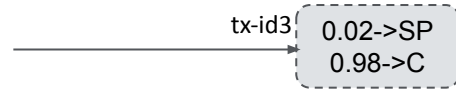


- (1) Service Provider (SP) gives their public key to Customer (C)
- (2) C creates tx-id1 which sends 0.1 BTC to a 2-of-2 address that unlocks with C and SP signatures; not broadcasted yet
- (3) C creates a refund tx-id2 that sends the funds back to C `nLockTime`'ed 12 blocks in the future (~2 hours) and sends to SP to sign
- (4) SP signs refund tx and returns
- (5) Then C broadcasts tx-id1 that sends the funds to the 2-of-2 address
- (6) C creates a tx-id3 which spends the 2-of-2 to send 0.01 to SP and 0.99 to C, signs and sends to SP (paying for 1 minute of service) —this is called a commitment tx and is off-chain
- (7) And again for each minute of streaming another commitment tx is made

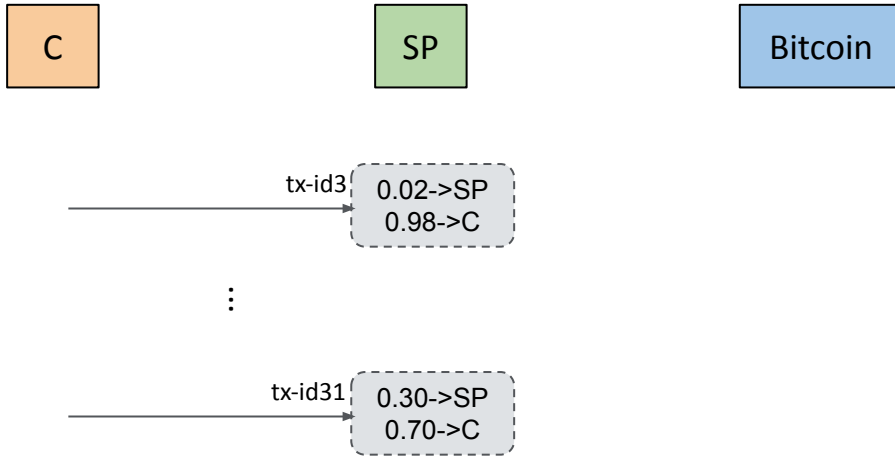
Spilman-style payment channels



(7) And again for each minute of streaming another commitment tx is made

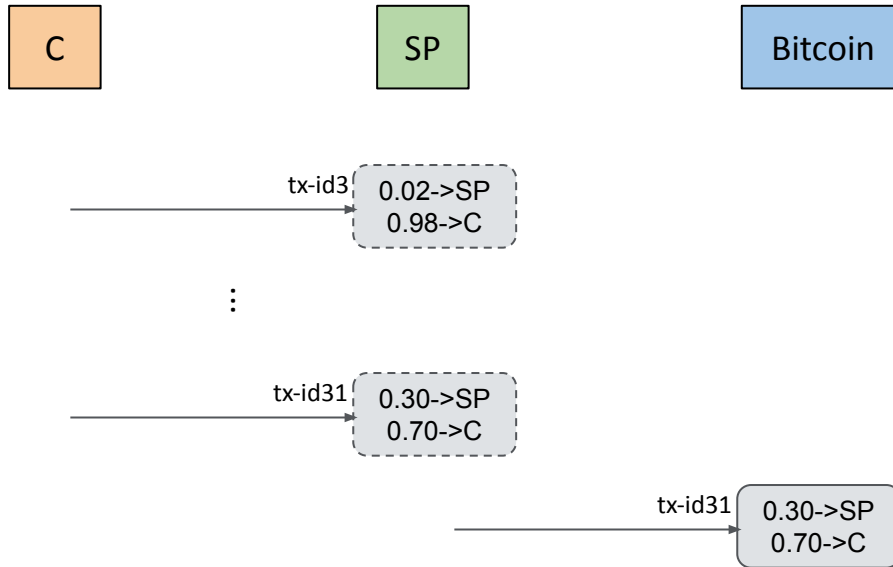


Spilman-style payment channels



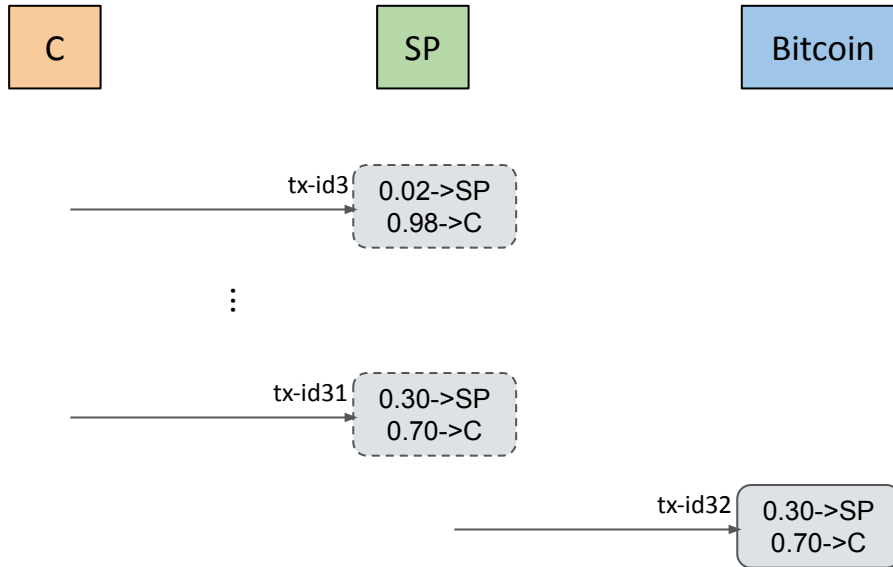
- (7) And again for each minute of streaming another commitment tx is made
- (8) And 28 more times...

Spilman-style payment channels



- (7) And again for each minute of streaming another commitment tx is made
- (8) And 28 more times...
- (9) C stops streaming service and SP collects the funds by signing and broadcasting the last commitment tx (`tx-id31`) to the network

Spilman-style payment channels



- (7) And again for each minute of streaming another commitment tx is made
- (8) And 28 more times...
- (9) C stops streaming service and SP collects the funds by signing and broadcasting the last commitment tx (tx-id31) to the network

NOTES:

- SP is incentivized to broadcast only the last tx.
- If SP does not cooperate C can broadcast the refund transaction after the `nLockTime` expires.

Spilman-style payment channels

Spilman-style payment channels are subject to transaction malleability attacks. For example after the refund transaction is signed and returned (step 4), the 2-of-2 transaction which is then broadcasted can be modified (step 5) before it is confirmed; the tx is acquired by the malicious party and the signature is slightly modified (still complying to DER, e.g. changing signs). This will change its txid which the refund tx depends on, invalidating the refund. Then the attacker re-broadcasts with a higher fee* so that miners choose it over the original one.

When setting up the refund tx we have to specify a future until which the refund cannot be claimed. This is the actual limit of the payment channel. It has to close before that time or else C can get a full refund. We could set this a long time in the future but then in case SP does not cooperate the funds will be locked accordingly.

Also notice that this kind of payment channels are unidirectional. Only one side pays the other.

* Or regardless of fees, the attacker arranges a deal with a miner which includes it in the next block.



CLTV-style payment channels

These are similar to Spilman-style channels but instead of creating an extra transaction for a refund you include the refund as a branch in the script of the funding address (i.e. the 2-of-2).

```
OP_IF
  2 <C Public Key> <SP Public Key> 2 CHECKMULTISIG
OP_ELSE
  <current block+12> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <C PKH>
  OP_EQUALVERIFY OP_CHECKSIG
OP_ENDIF
```

Now since the refund tx is part of the funding tx and not an additional dependent tx an attacker cannot use the tx malleability vulnerability previously described. This is an improvement of Spilman-style channels but they are still unidirectional and have a time limit.

However, the payment txs depend on the funding tx (possible malleability attack) and thus the SP is advised to wait for at least a single confirmation before accepting payments.



Poon-Dryja payment channels (punishment-based)

These payments are bidirectional and they have no time limit. With bidirectional channels both participants A and B can send funds to each other. The idea is similar to the previous examples where a funding tx opens the channel and then commitment txs change the balances. The funding tx however gets funds from both participants this time.

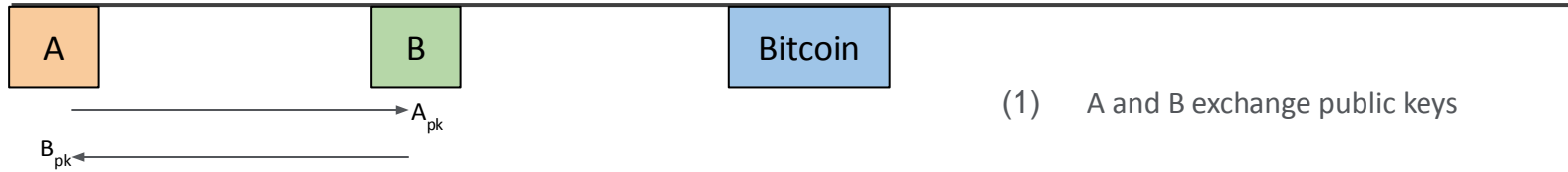
If one of the participant misbehaves the other one can actually claim all the funds back (punishing the bad actor).

Payment channels of this type became possible with Segregated Witness which solves the transaction malleability issues.

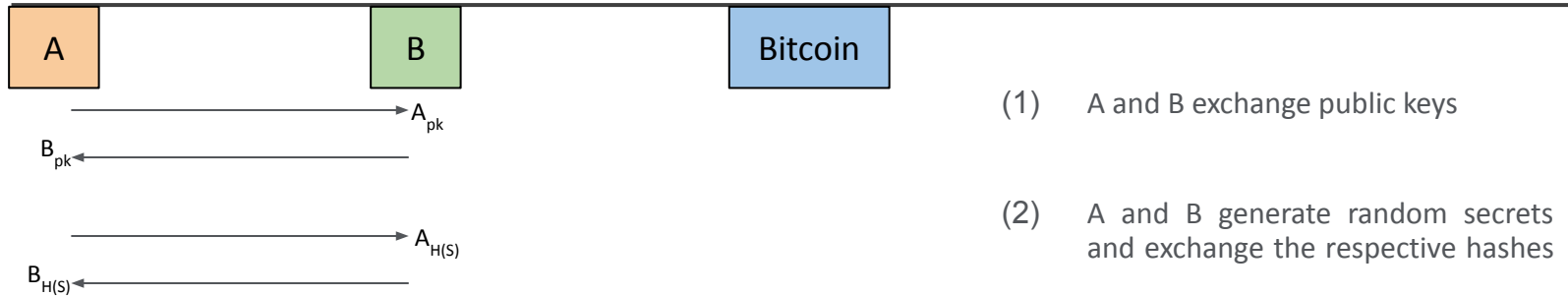
The variation that we will go through uses revocation keys, which demonstrates the basic idea applied in the Lightning Network.



Poon-Dryja payment channels



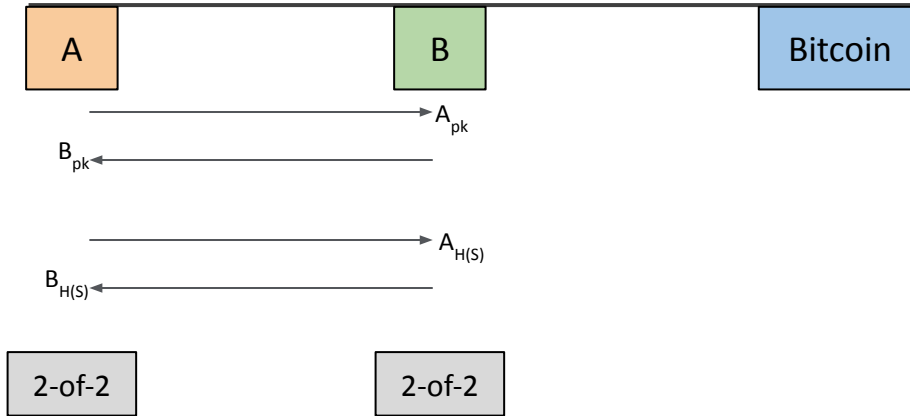
Poon-Dryja payment channels



(1) A and B exchange public keys

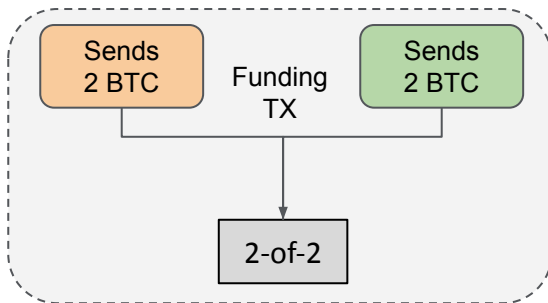
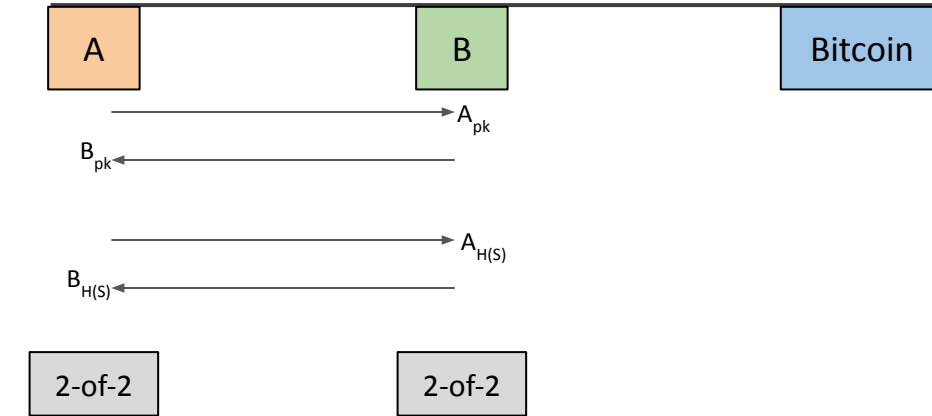
(2) A and B generate random secrets and exchange the respective hashes

Poon-Dryja payment channels



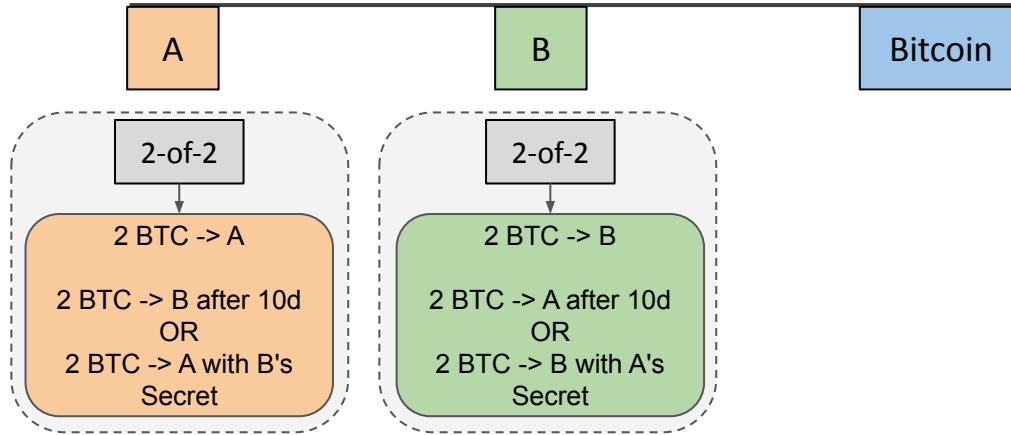
- (1) A and B exchange public keys
- (2) A and B generate random secrets and exchange the respective hashes
- (3) A and B create the 2-of-2 address controlled by A and B (they both know they public keys).

Poon-Dryja payment channels



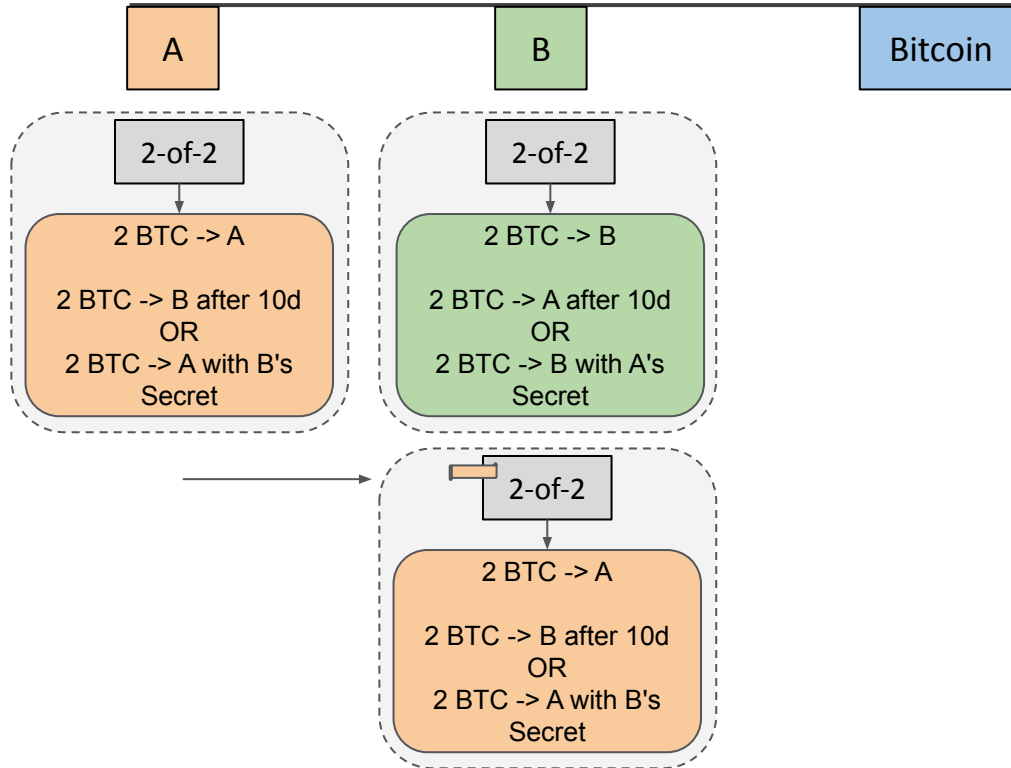
- (1) A and B exchange public keys
- (2) A and B generate random secrets and exchange the respective hashes
- (3) A and B create the 2-of-2 address controlled by A and B (they both know they public keys).
- (4) They both send the same amount of funds to that address but do not broadcast.

Poon-Dryja payment channels



- (5) Both A and B create a commitment transaction that return their funds immediately to them and the funds of their peer after some delay, say 10 days OR also to them if they know the other participant's secret.

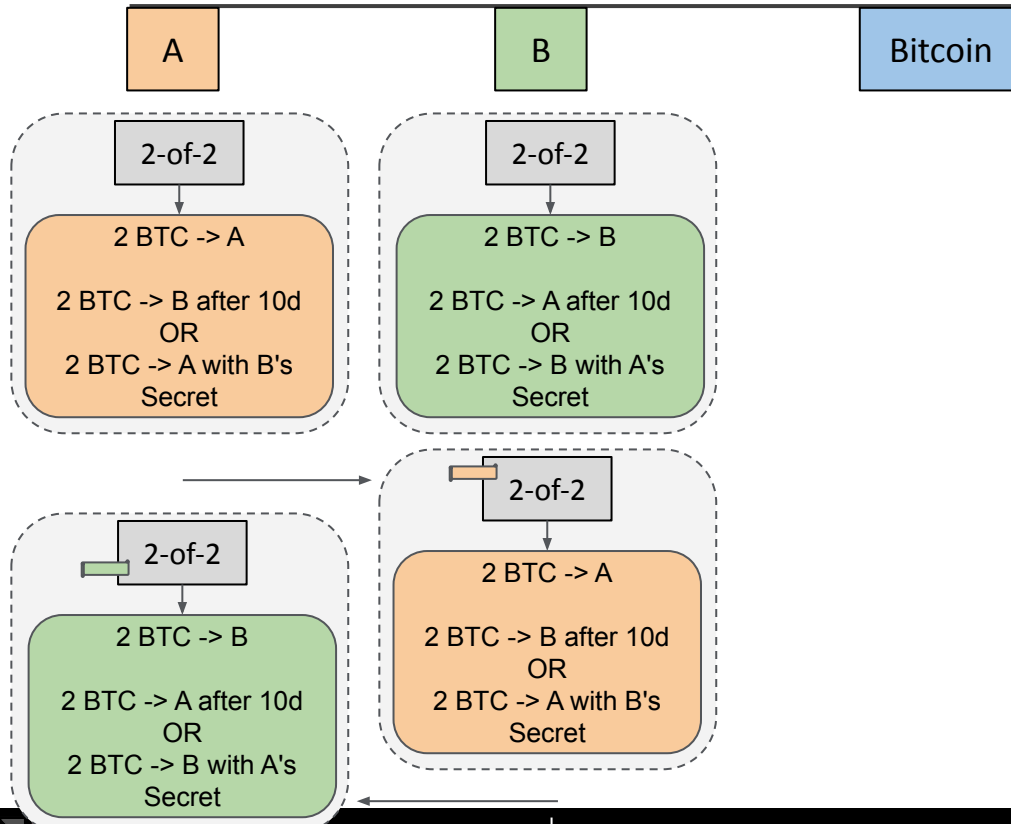
Poon-Dryja payment channels



- (5) Both A and B create a commitment transaction that return their funds immediately to them and the funds of their peer after some delay, say 10 days OR also to them if they know the other participant's secret.

- (6) A signs the 2-of-2 tx and sends to B. B can now sign and get refunded albeit with some delay.

Poon-Dryja payment channels

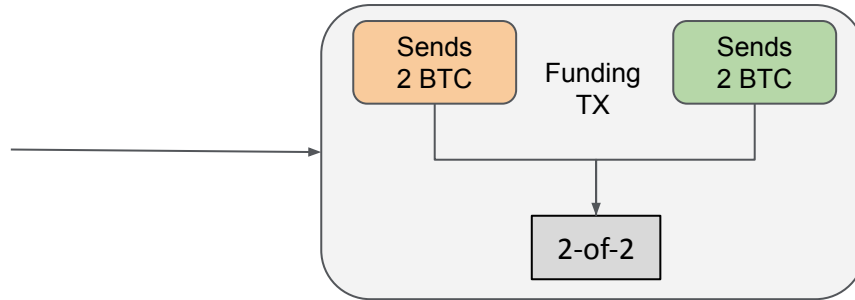


- (5) Both A and B create a commitment transaction that return their funds immediately to them and the funds of their peer after some delay, say 10 days OR also to them if they know the other participant's secret.
- (6) A signs the 2-of-2 tx and sends to B. B can now sign and get refunded albeit with some delay.
- (7) B signs the 2-of-2 tx and sends to A. A can now sign and get refunded albeit with some delay.

Poon-Dryja payment channels



(8) Now that both participants committed for refunds the funding transaction can be broadcast (channel is open).



Note: both A and B can now close the channel and claim their money back with a delay.

Poon-Dryja payment channels

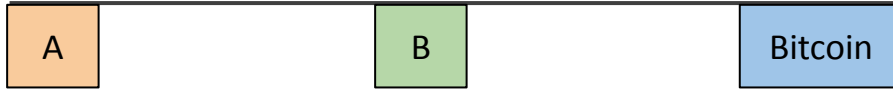


To use the channel participants have to create new commitment transactions (similar to above) to represent the new state of the funds.

Suppose B sends 1 BTC to A; the new state would be: 3 BTC goes to A and 1 BTC goes to B.

So the commitment process is repeated.

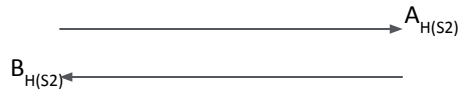
Poon-Dryja payment channels



To use the channel participants have to create new commitment transactions (similar to above) to represent the new state of the funds.

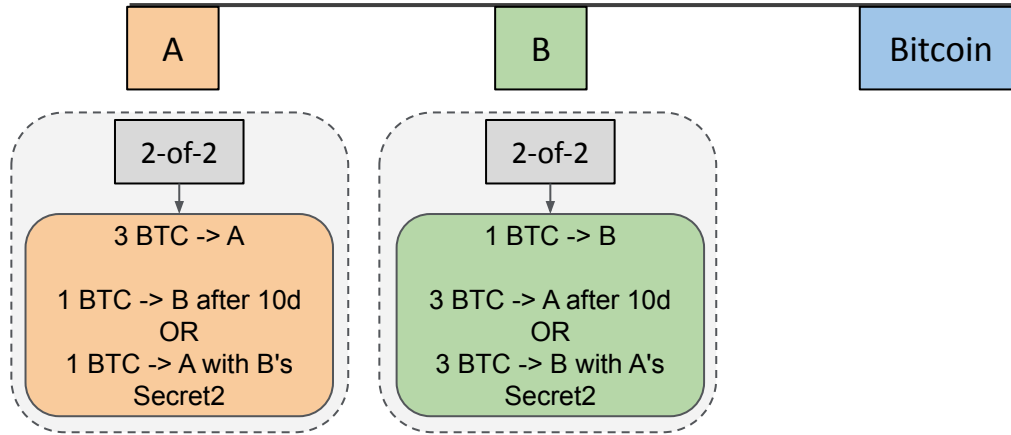
Suppose B sends 1 BTC to A; the new state would be: 3 BTC goes to A and 1 BTC goes to B.

So the commitment process is repeated.



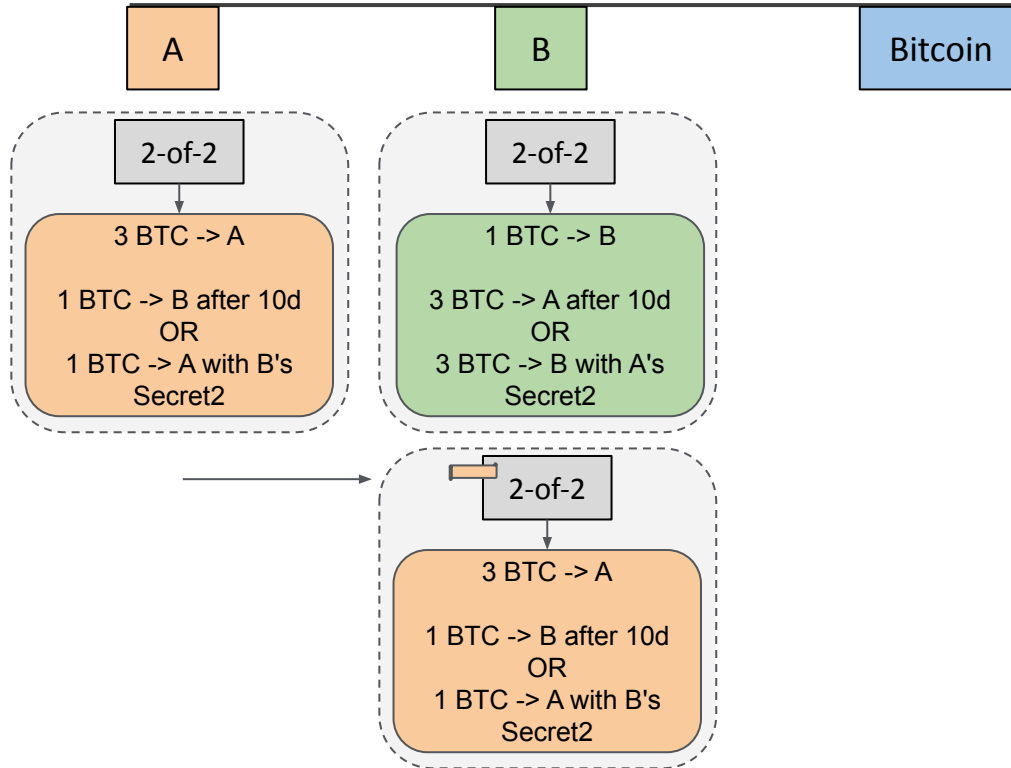
- (9) A and B generate new random secret (A_{S_2} and B_{S_2}) and exchange the respective hashes ($A_{H(S_2)}$ and $B_{H(S_2)}$)

Poon-Dryja payment channels



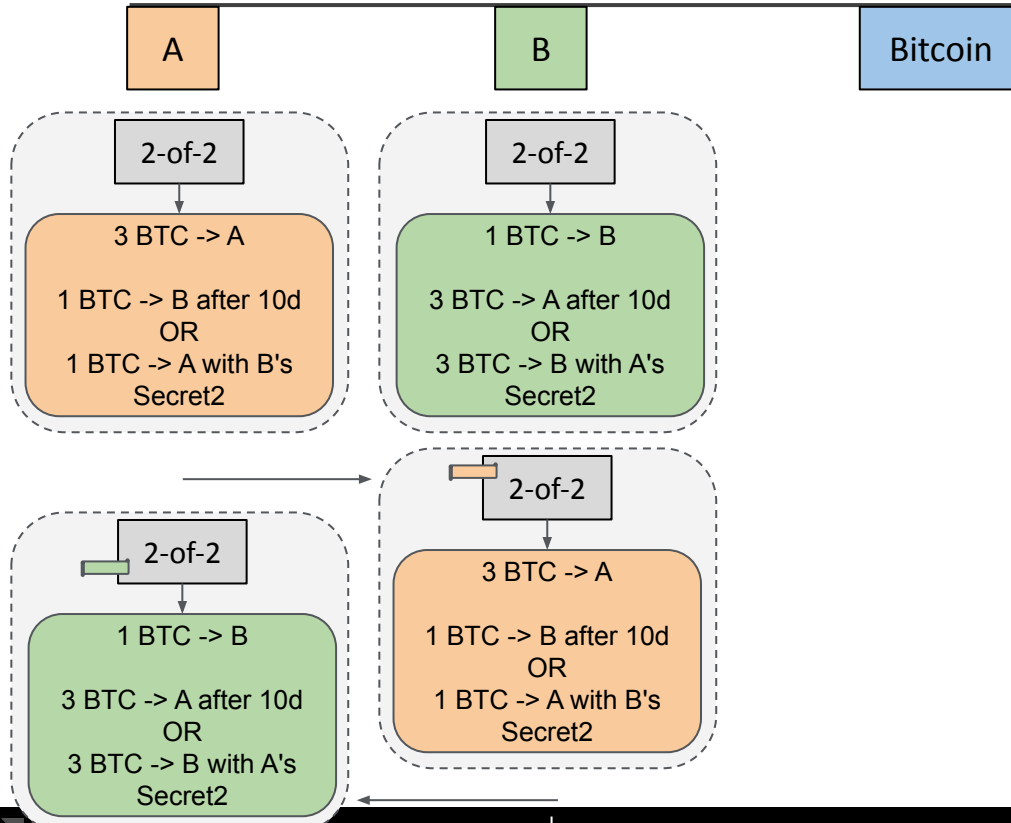
(10) B will use the channel to send 1 BTC to A. Both A and B create new commitment transactions that represent the new state of the funds; i.e. 3 BTC goes to A and 1 BTC goes to B.

Poon-Dryja payment channels



- (10) B will use the channel to send 1 BTC to A. Both A and B create new commitment transactions that represent the new state of the funds; i.e. 3 BTC goes to A and 1 BTC goes to B.
- (11) A signs the 2-of-2 tx and sends to B. B can now sign and get refunded albeit with some delay.

Poon-Dryja payment channels

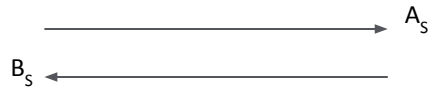


(10) B will use the channel to send 1 BTC to A. Both A and B create new commitment transactions that represent the new state of the funds; i.e. 3 BTC goes to A and 1 BTC goes to B.

(11) A signs the 2-of-2 tx and sends to B. B can now sign and get refunded albeit with some delay.

(12) B signs the 2-of-2 tx and sends to A. A can now sign and get refunded albeit with some delay.

Poon-Dryja payment channels



(13) Now the previous commitments need to be invalidated. We can exchange the previous secrets to accomplish that.

Poon-Dryja payment channels



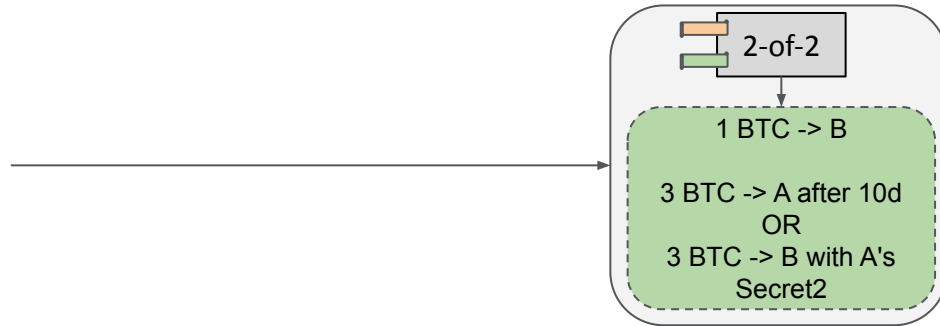
(13) Now the previous commitments need to be invalidated. We can exchange the previous secrets to accomplish that.

Effectively now we have moved to the new state. If B broadcasts an older beneficial commitment, e.g. from step 6, A has 10 days to use B's secret and claim B's funds from the second output (and can always claim the funds from the first output).

Poon-Dryja payment channels



(14) Participants can always broadcast the last state to close the channel. Here A also signs and broadcasts B's last commitment.

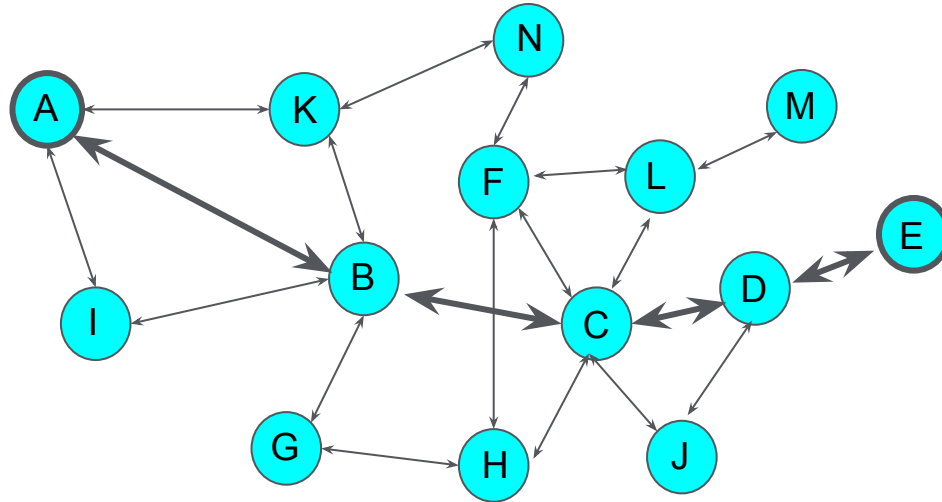


Section 2: Lightning Network



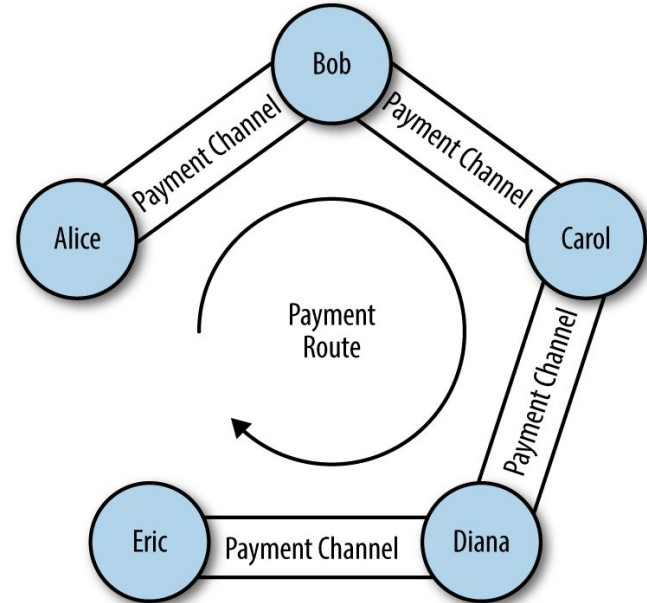
Lightning Network

It is not feasible to expect every user to create a payment channel with everybody that he needs to transact in some way. Lightning network is a design on how bidirectional payments channels can be routed securely among many participants. The idea is that user A will be able to pay user E without a direct payment channel. The payment would go through B, C and D.



Lightning Network

It was first described in the [lightning network paper](#) and it is *one* way that routed payment channels may work. There are several implementations and they are being guided by a set of interoperability documents called [Basics of Lightning Technology](#) or BOLT.



* Diagram from [Mastering Bitcoin](#) book, [Chapter 12](#)

Lightning Network

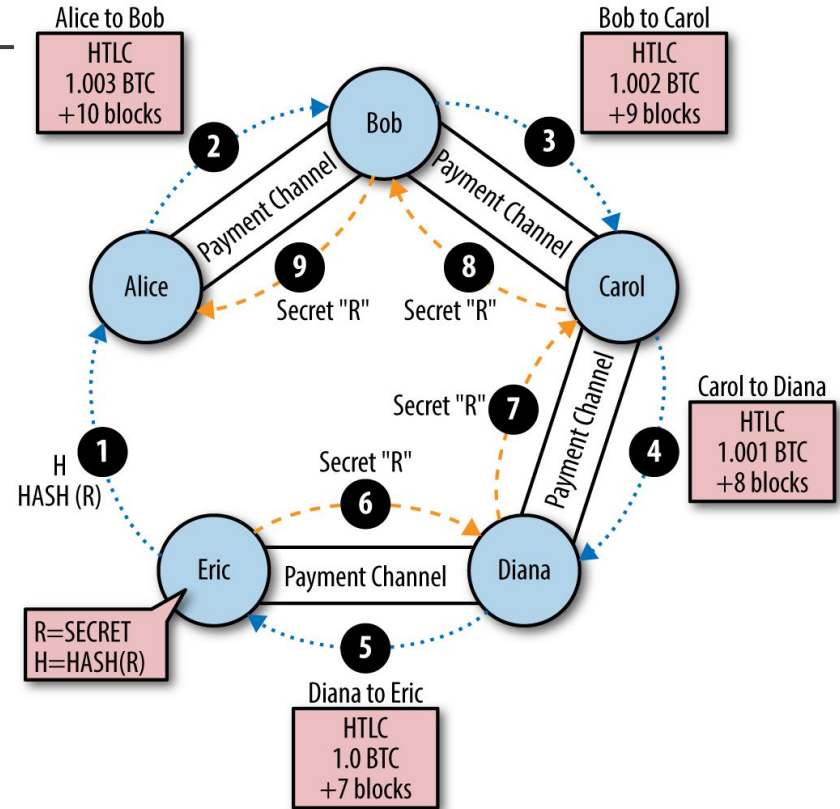
A routed payment channel example where Alice wishes to pay Eric 1 BTC. Eric constructs a secret and gives the hash to Alice.

The network allows Alice to know that Eric is connected to Diana, Diana to Carol, etc. and thus can construct the route.

Alice sends 1.003 BTC to Bob with a commitment stating that Bob will get paid after the secret is revealed.

The same happens until a commitment from Diana to Eric is sent (notice LN fees being reduced).

Eric gets 1 BTC by revealing the secret, which is then used by everyone else to get their 1 BTC.



* Diagram from [Mastering Bitcoin](#) book, [Chapter 12](#)

Lightning Network: Benefits and Challenges

Benefits:

Privacy/Fungibility: onion routing is used, i.e. each lightning node only sees the previous and the next node thus they do not know the sender or the recipient.

Speed/Capacity: the transaction commitments are off-chain and thus *lightning* fast and not restricted by on-chain block size constraints.

Granularity: micro-payments are possible

Challenges:

Limited routes and/or capacity: finding a route is not always possible or might take time to find an appropriate route; that is one of the primary critiques of lightning network. People argue that hubs will be required to enable better routing, which will decrease decentralization.



Conclusions



Conclusions

- We went through different approaches for Payment Channels and explained some of them in detail.
- Went through some of the state of the art developments of the Bitcoin network



Further Reading



Further Reading

Payment Channels

https://en.bitcoin.it/wiki/Payment_channels

<https://blog.chainside.net/understanding-payment-channels-4ab018be79d4>

Mastering Bitcoin (Ch.12), Andreas Antonopoulos

<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc>





UNIVERSITY *of*
NICOSIA