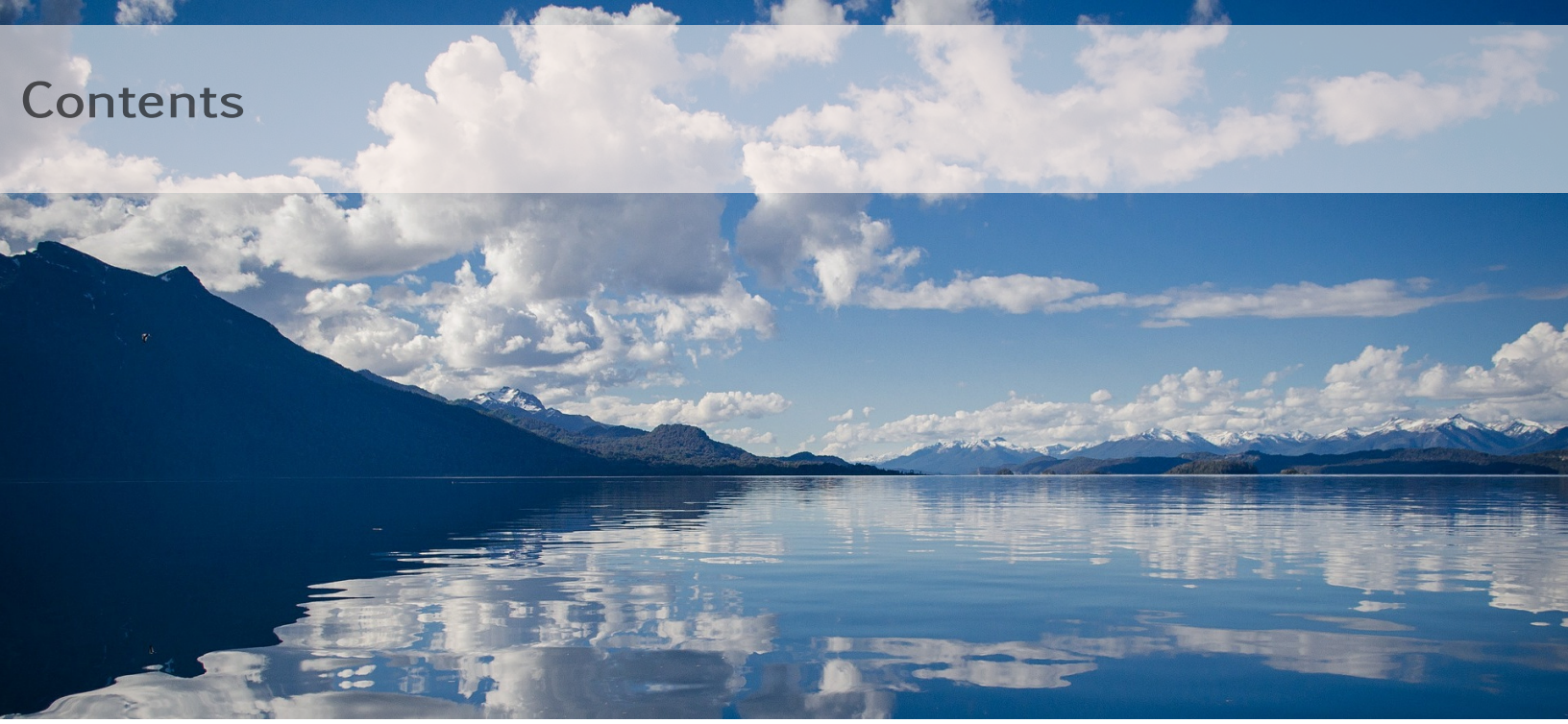
Bitcoin Programming

Konstantinos Karasavvas

v0.4

LaTeX template based on Typesetters (https://www.typesetters.se/latex-textbook-template/)

Contents



0.1 Preface

I started teaching Bitcoin programming in 2016. Every year I was trying to improve and update my material to keep it as relevant as possible. Luckily, Bitcoin progresses at a steady pace while always keeping backwards compatibility. This is convenient because existing material will always be valid even though better alternatives might be introduced in the future.

To understand the material better and to improve the material in my courses I started an open source Python library, called bitcoin-utils[1]. The library was created for educational purposes and not for computational efficiency and that might be evident in certain parts of the implementation. Before starting this library I had investigated several other well-known Python libraries but I did not find an appropriate one for teaching. Some were too low-level with limited documentation while others where abstracting concepts that I deemed where important for students to understand. The book contains a lot of examples using this library for demonstration.

Throughout the years I have prepared a lot of material based on my early code experiments, the bitcoin-utils library and several online resources, especially the Bitcoin Stack Exchange[2] and the Bitcoin Book[3] by A. Antonopoulos. While I try to always credit the initial sources that I have consulted over the years it is possible that I have missed some. Please let me know and I will update accordingly.

This book is not about introducting what Bitcoin[4] is. The readers are expected to have some programming knowledge, be comfortable with their own operating system as well as have some basic understanding of what Bitcoin is, its utility, what addresses and private keys are as well as have some experience using wallets and sending bitcoins[5]. However, the first chapters do give a quick summary of Bitcoin and how it works to make sure that the fundamentals are covered.

This book is about teaching Bitcoin programming; to help people delve deeper and in particular learn how to *talk* to the Bitcoin network programmatically. The Bitcoin library is relatively easy to navigate through, so you can go even deeper when required. There is an emphasis on providing practical examples which, I believe, helps understanding.

I have been using a Linux-based machine and thus most command-line examples are from **bash** shell. However, people comfortable with other Operating Systems should have no issues to adjust as needed.

[1] https://github.com/karask/python-bitcoin-utils

[2] https://bitcoin.stackexchange.com/

[3] https://github.com/bitcoinbook/bitcoinbook

[4] https://github.com/karask/satoshi-paper

[5] When we refer to the coins we will use *bitcoin(s)* (lowercase 'b') and when we refer to the protocol or the network we will use *Bitcoin* (uppercase 'B')

1. How Bitcoin Works

This chapter provides a high-level introduction of how Bitcoin works. It aims to be a summary of the prerequisite knowledge required by the reader before moving into the following chapters. The operation of the Bitcoin network is demonstrated with a walkthrough of a transaction and its journey from its creation up until its final destination, the Bitcoin blockchain.

1.1 The Story of a Transaction

Transactions specify the transfer of bitcoin ownership. Assume we have three actors; Zed, Alice and Bob. Zed has sent 1.5 bitcoins to Alice with TX_x and Alice wants to send 1 bitcoin to Bob. The transaction history will already have an entry of how Alice got her bitcoins (e.g. from Zed).

> Internally, the Bitcoin protocol operates with satoshis: $1\ satoshi = 0.00000001\ BTC$. Thus, when we want to transfer 1 BTC we actually transfer 100000000 *satoshis*.

To send 1 bitcoin (or *BTC*) Alice needs to 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 1 BTC from her 1Alice bitcoin address to Bob to his 1Bob address.

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

A transaction can consist of several *inputs* (outputs of past transactions) 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 contains as inputs need to be *spent*.

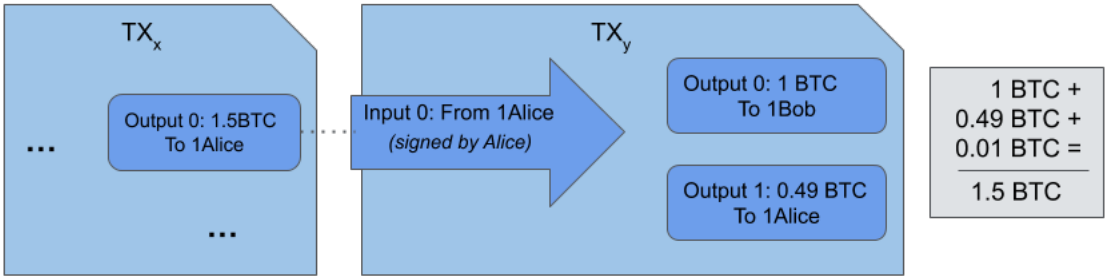


Figure 1.1: Typical one input two outputs transaction.

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 (see figure 1.1 where the miner receives 0.01 BTC). A typical transaction transfers some bitcoins to another user and returns the remaining bitcoins as change to the originating address or another address that the sender controls.

> For privacy reasons it is recommended to send the change to a different address than the originating. Most bitcoin wallets already do this behind the scenes.

Any number of inputs and outputs is possible as long as a transaction fee is included; the larger the transaction the larger the transaction fee. The unspent outputs are called *Unspent Transaction Outputs (UTXOs)* and the set of UTXOs is essentially all the available bitcoins in the network.

Once a transaction is created it needs to be sent to a Bitcoin node. After the node receives the transaction it checks if it is valid, e.g. the output amounts should be less or equal to the input amounts, the signature proving ownership should be valid, etc. If it is valid the node will propagate it to all its peers[1], i.e. the other nodes that it is aware of. In turn, the other nodes will check if the transaction is valid and so on and so forth until all nodes receive the transaction (see figure 1.2).

1.2 From Transactions to Blocks

From a Bitcoin's node perspective, the node receives transactions and places all valid ones into its memory pool, or *mempool*. It keeps receiving new ones until it decides that it will group some of those transactions into a block (see figure 1.3).

[1] To be more precise they will notify their peers of the transaction by its *transaction identifier (txid)* and the peers can choose to request it or not. More details will be provided in the Peer-to-Peer chapter.

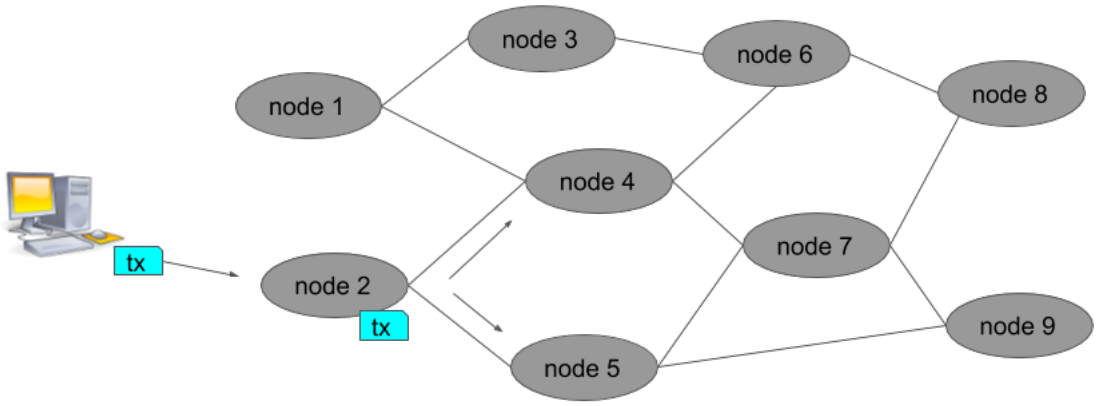


Figure 1.2: Example of transaction propagation through the network.

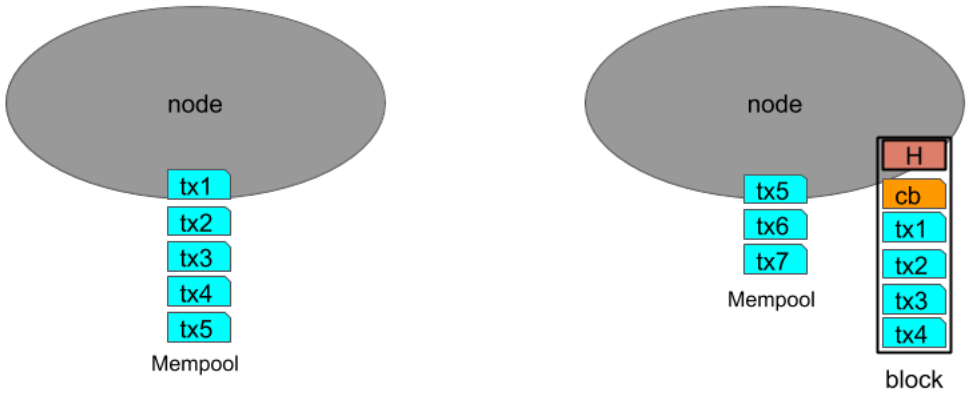


Figure 1.3: A node receives transactions into its mempool and can attempt to create new blocks for the network.

> We are describing what mining nodes typically do. The majority of nodes are not mining nodes and thus do not attempt to create new blocks, rather they validate and propagate valid transactions and blocks when they are aware of them.

Every block contains a *coinbase* transaction that is added by the miner (see next section) and it sends a deterministically calculated reward to an address of the miner's choosing. Finally a header is added to the block containing important information that links this block to its parent and other information that we will examine in the next section.

1.3 Mining: basics

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 (see figure 1.4). They can create and propagate a block at any time.

But how do we select which blocks will be part of the blockchain? Since, miners include a reward for themselves everyone wants their block to be the next block in the blockchain. In other words, how do we avoid spam and Denial of Service (DoS) attacks?

For a block to be considered valid a miner has to prove that he has done some intensive computational work. Thus, miners have to spend resources before they create a block. This mechanism of proving computational work is called *Proof-of-Work (PoW)* and it involves solving a problem or puzzle. PoW puzzles have the fundamental property of being difficult to solve but trivial to validate their correctness.

Bitcoin mining is the process of solving the PoW puzzle and selecting the next valid block in a way that is undisputed and thus achieve consensus on the current blockchain state. Bitcoin uses the Hashcash PoW algorithm [1] for its mining.

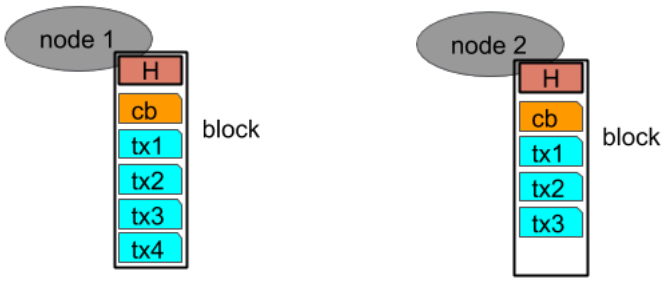


Figure 1.4: All nodes will eventually receive all transactions but they are free to include them into a block as they see fit.

The Proof-of-Work puzzle is to compute a cryptographic hash (effectively a big hexadecimal number) of the new block that we want to create which should be less than a target number. The target number that the hash needs to be less than can be deterministically calculated by all nodes and is such that it would take around 10 minutes to calculate with the current network processing power, also called hashing power. Since a hash is random it will take several attempts to find a proper hash but other nodes will verify with only one attempt.

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.

As more miners join the blocks will be created faster so the puzzle's difficulty automatically adjusts (increases) so that it again requires approximately 10 minutes to solve. This *difficulty adjustment* is happening every 2016 blocks, which is approximately 2 weeks if each block takes 10 minutes to mine.

The hash algorithm used is SHA256 and it is applied twice to the block header. As we will see later the header uniquely represents the whole block including all the transactions

and thus hashing the header is effectively the same as hashing the whole block, but much more efficiently since the header is much smaller.

```
SHA256( SHA256( block_header ) )
```

The miner that successfully creates a valid block first will get the bitcoin reward that they have set themselves in the coinbase transaction as well as the fees from all the transactions in the block.

The block reward can be deterministally calculated according to the current *block height*. The reward started at 50 bitcoins and is halved every 210000 blocks (approximately 4 years for 10 minute blocks). So, after block 630000 the reward will be 6.25 bitcoins. The mining reward can be claimed by the miner only after 100 *confirmations*, i.e. after 100 blocks have been confirmed as part of the blockchain since.

1.4 Mining: a bit more technical

The structure of the block header is as follows:

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

Block version and timestamp are self explanatory but we will briefly go through the remaining fields.

hashMerkleRoot

A block has two parts, the header and the transactions. 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 and being included in the block by **hashMerkleRoot**.

A merkle tree[2] 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.

An important property of a merkle tree is that you can efficiently prove that a hash (and thus transaction) is part of the merkle root. A merkle proof consists of the hashes required to reconstruct the merkle root from the leaf TX, thus proving that the TX hash is indeed part of the merkle tree. For example to prove that **tx1** is part of the merkle root we would need to provide the hash of **cb** and its positioning (i.e. left) as well as the parent hash of **tx2** and **tx3** (**cc3f1**) and its positioning (i.e. right).

[2] https://en.wikipedia.org/wiki/Merkle_tree

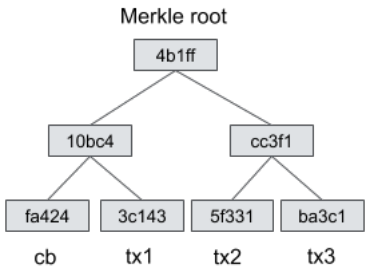


Figure 1.5: Simple merkle root calculation of coinbase and three transactions.

hashPrevBlock

This is the hash of the previous block in the blockchain. It designates the parent of the current block and it is effectively what chains the blocks together. For example, if someone changes a transaction in the previous block then the **hashPrevBlock** will change. This is of particular importance as we will see in more detail in section 1.5.

target

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:

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

The highest possible target (the easiest target) is defined as **0x1d00ffff** and gives a 32-byte target of (expressed as a 64 hexadecimal number):

```
0x00000000ffff0000000000000000000000000000000000000000000000000000
```

In Python that would be calculated as follows:

```
>>> 0x00ffff * 2**(8*(0x1d - 3))
26959535291011309493156476344723991336010898738574164086137773096960
>>> format(26959535291011309493156476344723991336010898738574164086137773096960, '064X')
'00000000FFFF0000000000000000000000000000000000000000000000000000'
```

Listing 1.1: Python examples

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

Another representation of target, easier for humans to understand, is *difficulty* which represents the ratio between the highest target and the current target:

$$Difficulty = \frac{highest\_target}{current\_target}$$

When Bitcoin started it started with the highest target (**0x1d00fff**) and thus it had difficulty 1. Difficulty 1 requires 2^{32} attempts on average to find a solution. Difficulty 10 requires $2^{32} * 10$ attempts on average, etc.

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 4.2 billion combinations, while we need quintillion nowadays[3].

When the limit was reached miners started modifying the timestamp (e.g. -1 sec) to allow for an additional of 4.2 billion 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[4].

Finally, miners started to use the unused space of coinbase's transaction input as an extra nonce allowing an immense amount of extra nonces to be used[5].

Difficulty Adjustment

We already mentioned that 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. This can be seen in figure 1.6.

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:

$$old\_difficulty * (\frac{2\_weeks}{t})$$

Mining Process in a nutshell

Here we describe a mining node's actions for mining in a simplified step by step process:

1. Gather valid TXs into blocks

[3] As of August 2020.

[4] The timestamp must be higher than the median of the 11 immediate ancestors of the block and higher than the timestamp of its parent. Finally, the timestamp must be no more than 2 hours in the future.

[5] The size of the coinbase input can be from 2 to 100 bytes.

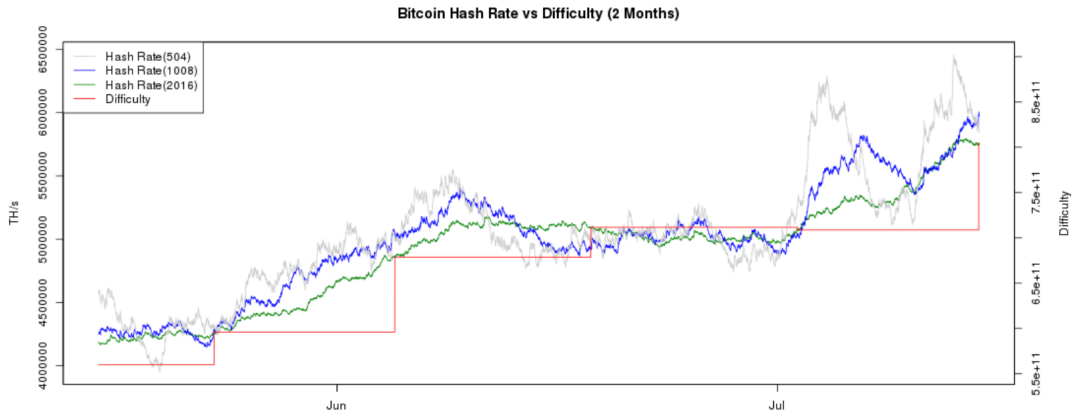


Figure 1.6: Hashrate and difficulty adjustment.

2. Get the longest chain's top block hash and add it in hashPrevBlock
3. Add timestamp, nonce and extra nonce in the first TX (coinbase)
4. Calculate the merkle root of valid TXs and add it to hashMerkleRoot
5. 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

During the above process:

- 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
    - update the timestamp
    - recalculate the merkle root
    - use the new block as hashPrevBlock

1.5 The story of a Block

Once a node finds a solution to the PoW problem it will propagate it to its peers. They will check if the block (and every transaction) is valid and if it is they will propagate it to all their peers[6]. In turn, the peers will check again for the solution as well as the block validity and they will propagate again and so on and so forth until all nodes receive the new block (see figure 1.7).

[6]To be more precise they will notify their peers of the block and the peers can choose to request the actual block or not. More details will be provided in the Peer-to-Peer chapter.

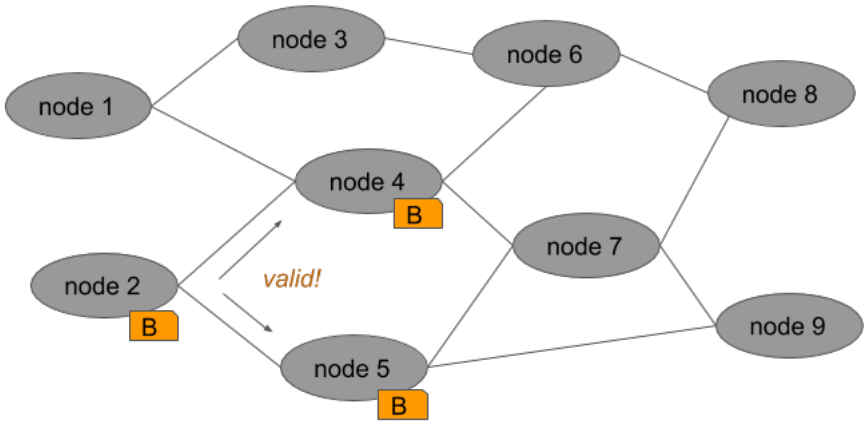


Figure 1.7: Example of block propagation through the network.

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 (see figure 1.8. The more confirmations the more final and secure a transaction is.

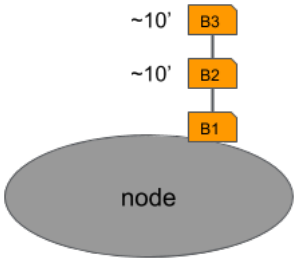


Figure 1.8: A node receives blocks and links them to form the blockchain.

1.6 Nakamoto Consensus and Trust

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. For an example see the different steps in figure 1.9.

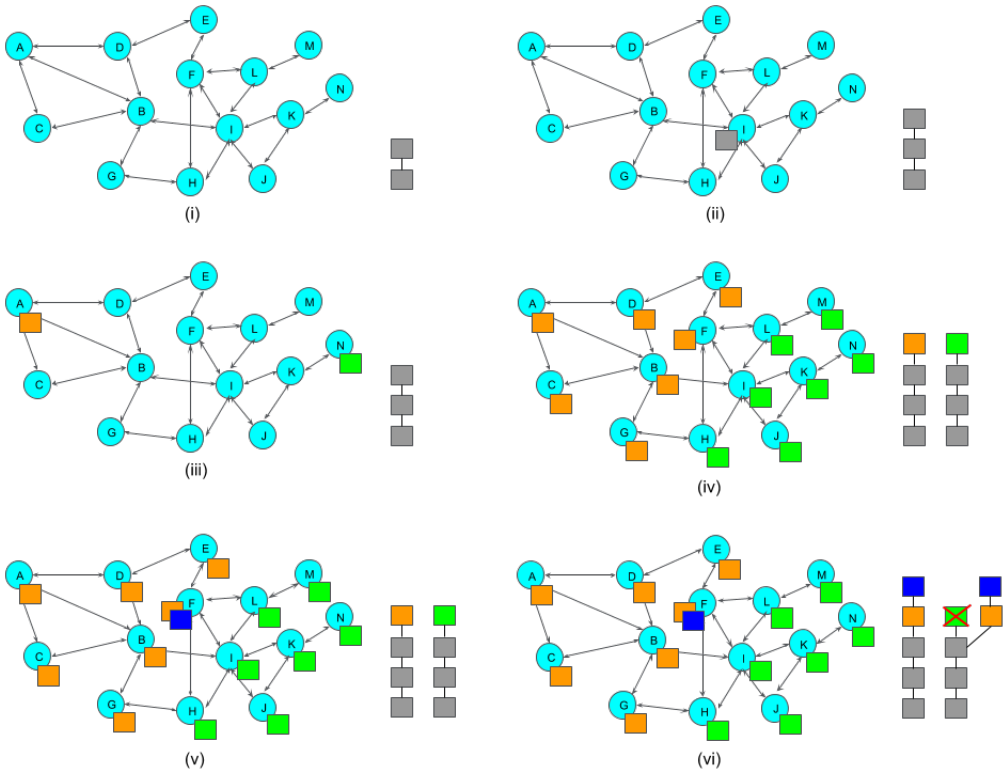


Figure 1.9: Nakamoto Consensus example.

(i) Initially our example network has only two blocks and all nodes are in sync.

(ii) Then node **I** finds the next block, which is disseminated to all other nodes and again everyone is in sync.

(iii) Next, let's suppose that two nodes find a solution at about the same time. These blocks will typically be very similar, including almost the same transactions, but will be different.

(iv) The nodes will propagate their blocks and some peers will get one of the blocks and some the other. The nodes are aware of both blocks but they will use the block that they received first as the next block and will start mining the next block on top of that block. At this stage we do not have consensus since some of the nodes have a blockchain with the orange block at the top and some the green block.

(v) However, after a while a new block will be found. In our example, this is node **F** and it will propagate it throughout the network.

(vi) Finally, when the block is propagated to the nodes with the green block on top they will realise that there is a longer chain than the one that they are working on. According to nakamoto consensus the nodes will accept the longer chain as the *valid* chain and ignore the green block. The green block is typically called an *orphan* block[7]. And now

[7] Precise terminology is more complicated as discussed in https://bitcoin.stackexchange.com/questions/5859/what-are-orphaned-and-stale-blocks

all nodes are in sync again.

> If there are transactions in the orphaned block that are not in the orange or the blue block they are moved to the mempool ready to be included in one of the following blocks.

Of course it would be possible that at step (v) 2 new solutions could again be found one from a node with an orange block on top and another from a green block on top. Similarly, consenus would be achieved with the following block and two blocks would be orphaned. In such a case, we would say that a 2-block *reorg* occurred.

Nakamoto consensus is a natural and expected reorg event that currently occurs less than once per month[8] even for 1-block reorgs. The chance of a reorg is proportional with the number of blocks and thus larger reorgs are exceedingly rare.

Establishing Trust

As we have seen, blocks are linked together by including the hash of the previous block on the new block. For example, in figure 1.10 the hash of **B1** is included in the header of **B2**.

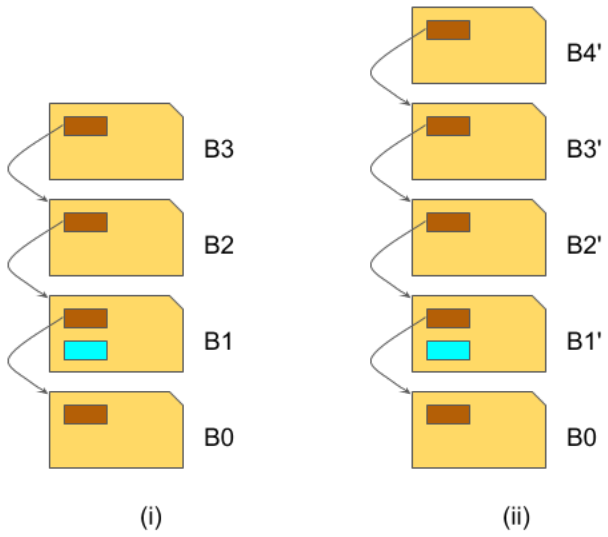


Figure 1.10: Linked blocks and Trust.

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[9] they will need to create and new **B1'** block with the modified transaction. However, there are two more blocks on top of **B1** and

[8] https://bitcoinchain.com/block_explorer/orphaned
[9] https://en.wikipedia.org/wiki/Double-spending

thus the attackers 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.

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[10].

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.

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 securing hundreds of billions worth of value. The Bitcoin blockchain is considered by many as the most immutable structure constructed by humans.

1.7 Basic interaction with a node

After installing the Bitcoin software[11] we can notice that it includes several executables, one providing the core functionality and the other for interaction and extra utility:

bitcoind: The daemon server that implements the Bitcoin protocol and networking functionality. It also includes a wallet. It provides a JSON-RPC API to talk to the node (ports: mainnet: 8332, testnet: 18332, regtest: 18443, sigtest: 38332).

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 Allows to create, parse or modify transactions.

[10] Certain attacks, like selfish mining, could be successful with a smaller percentage.

[11] You can find several online resources on how to install Bitcoin specifically for your operating system. Such tutorial is outside the scope of this book.

Bitcoin software configuration and development environments

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). Some important options for development and testing[12] your application include:

deamon=1 Runs the Bitcoin node in the background.

server=1 Allows JSON-RPC commands but only from localhost.

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. Available from Bitcoin Core v0.21.0.

addnode=12.34.56.78 Also connect to specific peer (multiple **addnode**'s can be used). If no network is specified it will only apply to mainnet.

connect=12.34.56.78 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).

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

mempoolsize=100 Only keep transactions that fit in 100 MiB. Transactions are ordered by fee rate and if there is not enough space the ones with the lowest fee rate are removed.

We can also include sections like **main**, **test** and **regtest** to provide specific options depending on the network used. Usually, when optioins are not specified under a section they apply to all sections with the exception of **addnode**, **connect**, **port**, **bind**, **rpcport**, **rpcbind** and **wallet**.

A typical minimalistic config for development is:

```
daemon=1
testnet=1
#regtest=1

server=1
rpcuser=kostas
rpcpassword=toodifficulttoguess
```

[12] https://developer.bitcoin.org/examples/testing.html

```
[main]
mempoolsize=300

[test]
mempoolsize=100

[regtest]
mempoolsize=20
```

Examples of Calls using bitcoin-cli

To get help of all available commands and how to get further help:

```
$ bitcoin-cli help
```

After version 0.22 a bitcoin wallet is not created by default. We have to create one ourselves. We can create a default wallet with:

```
$ bitcoin-cli createwallet "testwallet" load_on_startup=true
```

The default wallet is a descriptor wallet (see chapter 5 for more on descriptor wallets). To use commands like **importaddress** and **dumpprivkey** a non-descriptor wallet needs to be created:

```
$ bitcoin-cli createwallet "testwallet-2" descriptors=false
```

To get the current block height:

```
$ bitcoin-cli getblockcount
1806981
```

To get the current balance of all addresses:

```
$ bitcoin-cli getbalance
1.51815479
```

To get a new legacy[13] address:

```
$ bitcoin-cli getnewaddress "" legacy
mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS
```

By default Bitcoin v0.20+ use **bech32** (or native segwit) addresses. The example above overrides the default. We will examine the different kind of Bitcoin addresses in Chapter 5.

To encrypt the wallet with a passphrase:

```
$ bitcoin-cli walletencrypt PaSsPhRaSe
wallet encrypted; Bitcoin server stopping, restart to run with encrypted
wallet. The keypool has been flushed and a new HD seed was generated (if
you are using HD). You need to make a new backup.
```

To unlock an encrypted wallet for 2 minutes to spend funds:

```
$ bitcoin-cli walletpassphrase PaSsPhRaSe 120
```

To create a wallet backup:

```
$ bitcoin-cli backupwallet wallet.backup
```

To import a backed up wallet:

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

[13] The typical P2PKH Addresses starting with **1** on mainnet and **m** or **n** on testnet.

To get the node's networking info:

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

To get the node's blockchain info:

```
$ bitcoin-cli getblockchaininfo
{
  "chain": "test",
  "blocks": 1887283,
  "headers": 1887283,
  "Bestblockhash": "0000000000074e...9d44e05b4",
  "difficulty": 1420477.254893854,
  "mediantime": 1604662239,
  "verificationprogress": 0.9999999194957088,
  "initialblockdownload": false,
  "chainwork": "000000000000...a2762e8",
  "size_on_disk": 28640545955,
  "pruned": false,
  ...
}
```

To get the node's mining info:

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

To get the node's wallet info:

```
$ bitcoin-cli getwalletinfo
{
  "walletname": "wallet-test1",
  "walletversion": 169900,
  "format": "bdb",
  "balance": 0.00065838,
  "unconfirmed_balance": 0.00000000,
  "immature_balance": 0.00000000,
  "txcount": 1,
  "keypoololdest": 1667229693,
  "keypoolsize": 1000,
  "hdseedid": "c41632b90911bb4d4b172190bf9a27def9535fc4",
  "keypoolsize_hd_internal": 1000,
  "paytxfee": 0.00000000,
  "private_keys_enabled": true,
  "avoid_reuse": false,
  "scanning": false,
  "descriptors": false,
  "external_signer": false
}
```

To send 0.01 BTC to address **mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS** without specifying which UTXOs to use[14]:

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

To display all UTXOs with at least 2 confirmations:

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

[14] You can check the transaction online using a Bitcoin testnet block explorer: https://blockstream.info/testnet/tx/ff8322626c21c5bdfa1d27f75a55a1cb1d3b764bb34063f64b38f0803c370c08

To check all the available address labels:

```
$ bitcoin-cli listlabels
{
  "": 1.01483854,
  ...
}
```

To check all addresses with a particular label:

```
$ bitcoin-cli getaddressesbylabel
{
  "mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS": {
    "purpose": "receive"
  },
  ...
}
```

To get more information for the status of your node you can use commands like: **getblockchaininfo**, **getmempoolinfo**, **gettxoutsetinfo**, **getmemoryinfo**, **getrpcinfo**, **getmininginfo**, **getnetworkinfo**, **getpeerinfo**, **getdescriptorinfo**, **getaddressinfo**, **getwalletinfo**.

1.8 What to read next?

We should now have a basic understanding of how Bitcoin works and how to interact with a node. Make sure you are comfortable with the command-line, use **help** to see what commands are available and experiment!

After that you have some options on how to proceed. If you want some more background knowledge of how the Bitcoin network operates, continue with chapters 2 and 3. If you want to go straight to how to create transactions and write Bitcoin scripts programmatically go directly to chapters 5 and 6. For those who want to go deeper there is also the option to go through the implementation of the **bitcoin-utils** library. To that end chapter 4 will provide some technical knowledge required for understanding the library.

1.9 Exercises

Exercise 1.1 Prepare a bitcoin environment by installing a Bitcoin node configured for testnet.

Exercise 1.2 Using **bitcoin-cli** create a new legacy address.

Exercise 1.3 Use a Bitcoin testnet faucet to get some testnet bitcoins (tBTC) to one of your addresses.

Exercise 1.4 Ask your classmates or friends for their testnet address and send them some tBTC using **bitcoin-cli**.

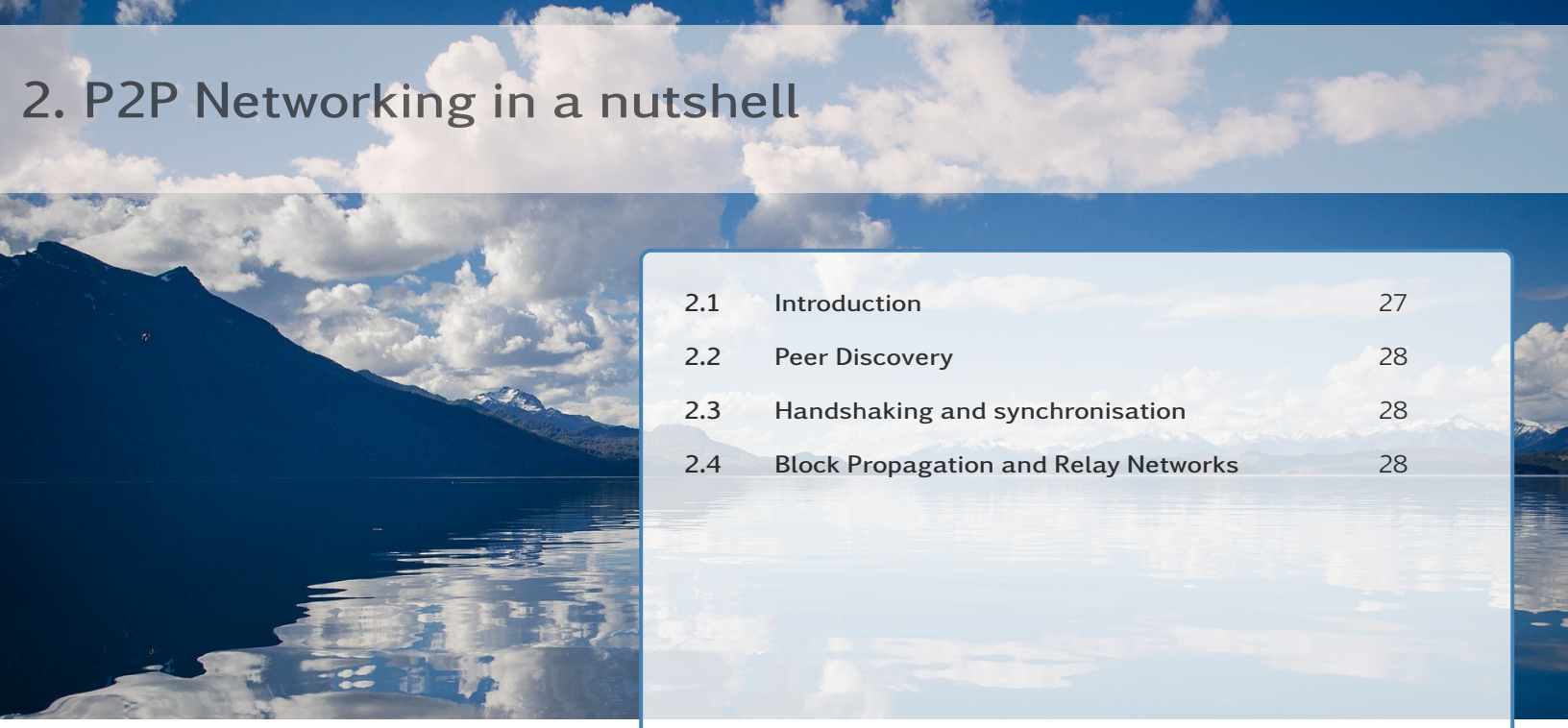
Exercise 1.5 Use a block explorer to see the status of the transaction that you created in the previous exercise.

Exercise 1.6 Encrypt your wallet and back it up.

Exercise 1.7 Go through the rest of the API and get familiar with more commands.

Exercise 1.8 Search for historical data on Bitcoin's difficulty adjustments and make sure you understand what you see.

2. P2P Networking in a nutshell



This chapter aims to introduce the very basics of the Bitcoin Peer-to-Peer networking, discussing peer (or node) discovery, blockchain synchronisation and others.

2.1 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 (irrespectively of its use[1]) 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[2]. Finally, only a few of those will provide mining services.

In November 2022 there were around 14663 nodes[3] in the network with ~99.9% of them using the Bitcoin Core implementation while the rest consists of alternatives like Bcoin, Bitcoin Unlimited, Bitcoin ABC, etc. A significant number of nodes (~53%) were using the Tor network[4]. With regard to mining there were 15 known mining pools[5] and other unknown ones corresponding to the mining Tor nodes.

[1] Wallet functionality can be disabled with **disablewallet=1** in **bitcoin.conf**. After v0.22 a wallet is not created by default.

[2] This is accomplished by specifying, say, **prone=1000** in **bitcoin.conf** to keep only the latest 1000 MiB of blocks

[3] https://bitnodes.io/

[4] https://www.torproject.org/

[5] https://www.blockchain.com/charts/pools

2.2 Peer 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** network 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.

2.3 Handshaking and synchronisation

When a node connects to a new peer it initiates a *handshake* by sending a **version** network 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 the blockchain from its peers. This initial synchronisation is called **Initial Block Download** or **IBD**. It sends a **getblocks** message which contains 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.

2.4 Block Propagation and Relay Networks

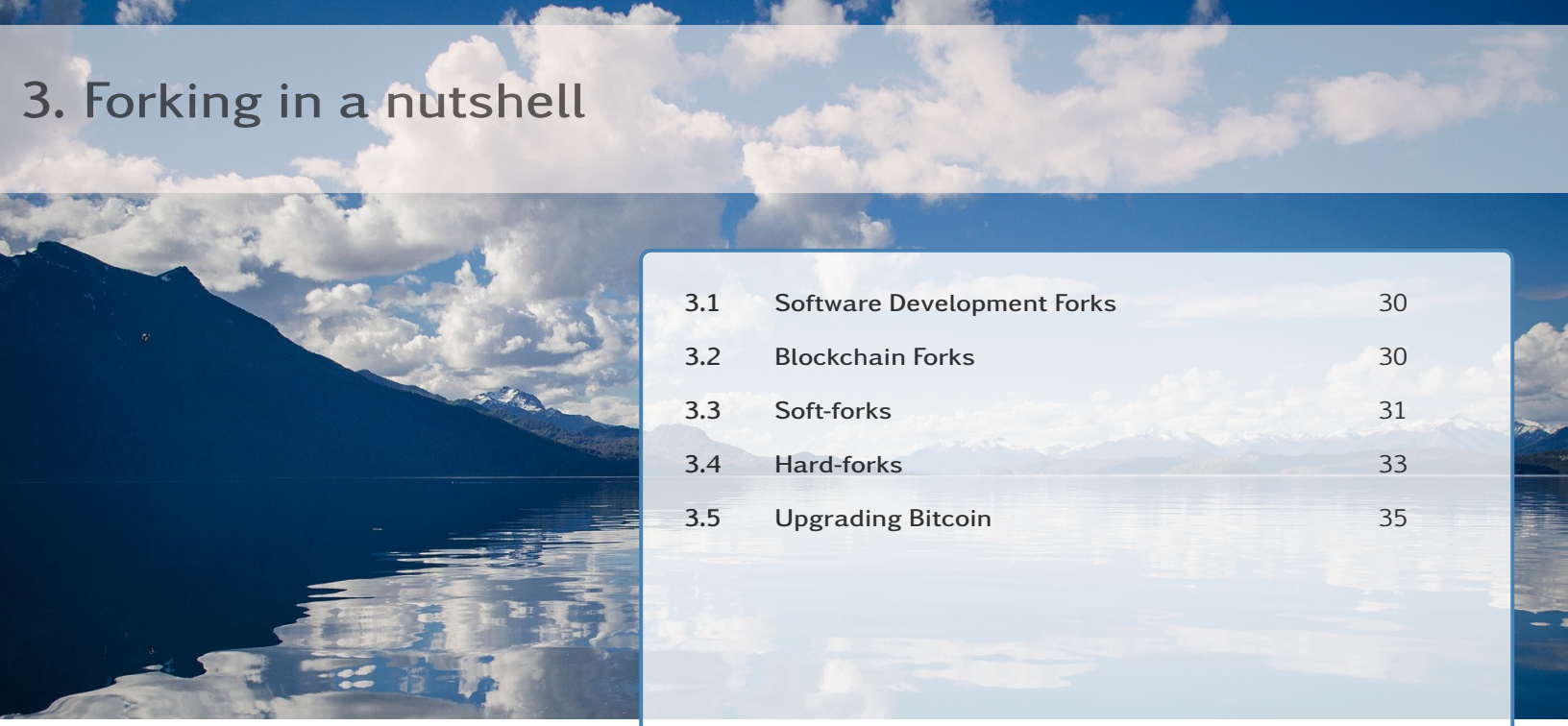
The faster a miner receives a new block the faster they can start working on the next block. Network latency is extremely important and since the P2P network takes some time (at least for the miners' needs) there are specialized networks to help with block propagation, for example the *Fast Internet Bitcoin Relay Engine (FIBRE)*[6].

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 block rates.

[6] http://bitcoinfibre.org

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.

3. Forking in a nutshell



This chapter will look into the process of upgrading the network. It explains what forking the blockchain means and what are the consequences.

3.1 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 (and Ubuntu from Debian)
- MariaDB from MySQL
- PostgreSQL fron Ingres
- OpenSSH from OSSH
- Inkscape from Sodipodi (and Sodipodi from Gill)
- Plex from XBMC
- And many many more.

3.2 Blockchain Forks

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 implemented rules 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.

Since running new code might result in a network (aka chain) fork the only way to update the Bitcoin protocol[1] is by forking.

> Temporary branching on the blockchain can sometimes occur and it is part of the Nakamoto consensus. Forking refers to compatibility breaking changes between peers.

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. Bitcoin Core v0.12.0 and Bitcoin Classic v0.12.0. The two groups will have different visions on how they wish Bitcoin to evolve and thus compete to gain the majority of the hashing power in order for their chain to prevail[2].

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

3.3 Soft-forks

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 backward 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.

A easy to understand example would be a new rule that decreases the maximum block size to 300kB. From now on new nodes will accept as valid only blocks that are 300kB or less. These blocks are also valid to the old nodes so it is a soft-fork change. Since the new blocks

[1] In particular the consensus rules.

[2] Prevail in terms of hashing power. Other than that both networks can co-exist.

have the majority of the hashrate the chain will eventually consist of only 300kB blocks and old nodes will slowly upgrade to the new rules[3].

Another example is Segwit but that involved several changes. Several sophisticated modifications were required to implement it a soft-fork; see section 6.5 for more details.

To summarize, in a typical soft-fork, the new rules do not clash with the old rules. For a step by step example of a soft-fork and how new blocks are added on top see figure 3.1.

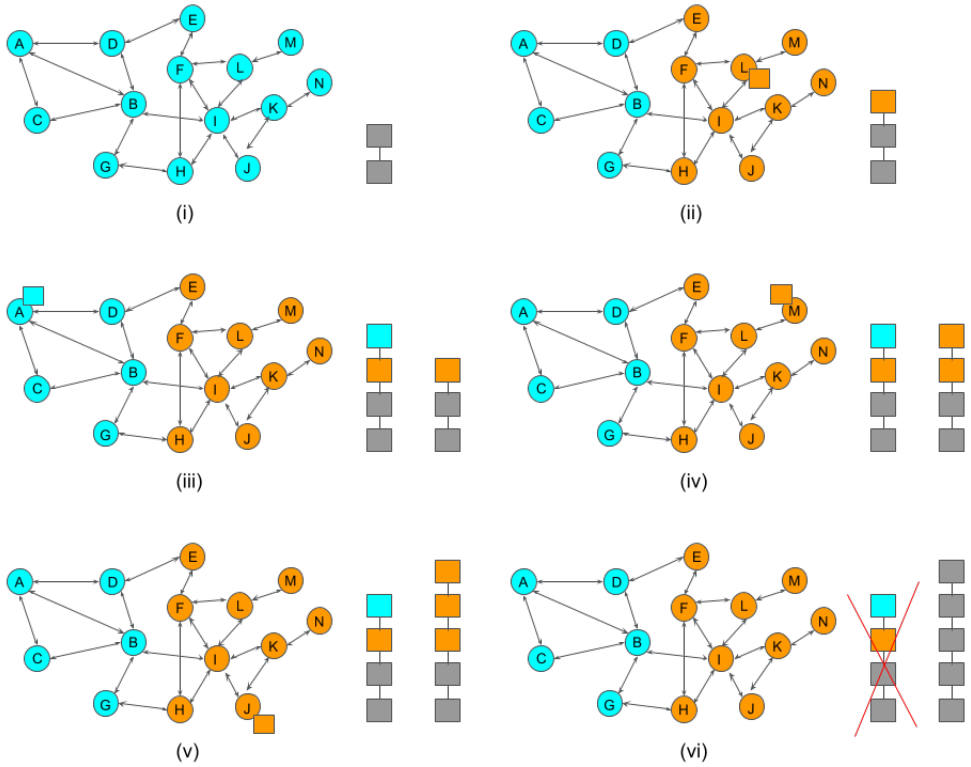


Figure 3.1: A soft-fork example.

(i) Initially our example network has only two blocks and all nodes are in sync.

(ii) Then, a soft-fork upgrade occurs where 67% of the network uses the new rules and one of the new nodes finds a new block. That block is accepted by everyone since new rules are a subset of the old rules. All the nodes are still in sync.

(iii) Next, a block with the old ruleset is created. It is only accepted by old nodes and thus we now have a temporary split.

(iv) Next another block with the new rules is found. It is accepted only by the new rules since the chains' tips are different. Note that the chains are now of equal size.

(v) Then yet another block with the new rules is found (67% chance!). Again, it is added on top of the chain with an orange block on top. However, the chain with the new rules is now the longest chain!

[3]They don't have to upgrade but their larger than 300kB blocks will never be finalized in the blockchain, so they might as well upgrade.

(vi) The old nodes are forced to follow the longest chain and thus the network is in sync again.

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.

> Whenever a longer alternative chain appears nodes have to accept it and substitute their latest blocks up until a common parent is found with the new chain. When that happers we say that a re-organization or *reorg* occured.

3.4 Hard-forks

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[4].

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.

For a step by step example of a hard-fork and how new blocks are added on top see figure 3.2.

(i) Initially our example network has only two blocks and all nodes are in sync.

(ii) Then, a soft-fork upgrade occurs where 67% of the network uses the new rules and one of the new nodes finds a new block. The new block is accepted only by the nodes with the new rules and thus we have a split.

(iii) Next a block is found based on the old rules which is accepted only by the nodes running the old rules.

(iv) Finally, another block is found by the nodes running the new rules which goes on the respective chain. The old nodes will never accept the incompatible blocks even from a longer chain. The split is permanent.

If a hard-fork occurs the network is effectively split in two. The mining hashrate is split in two as are the merchants and users. If one side does not change their software a hard-fork

[4]Of course this assumes that the new incompatible feature is being used in that block.

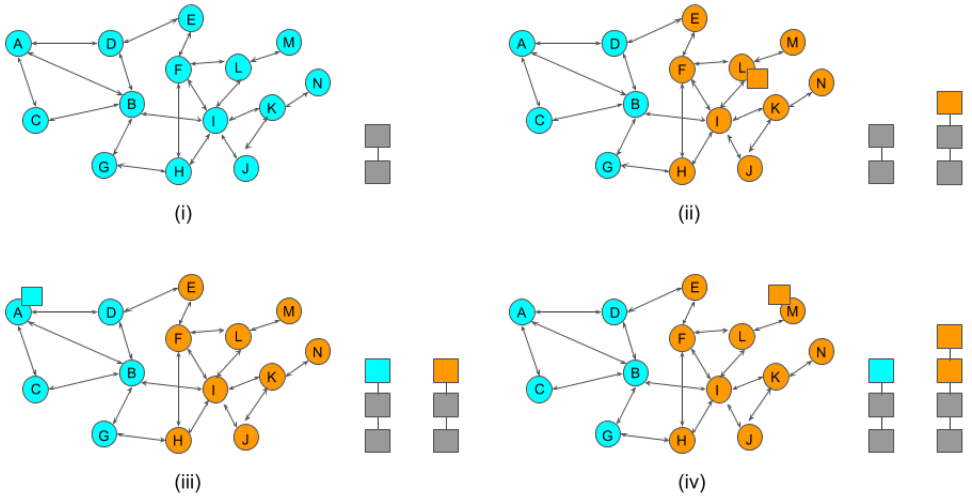


Figure 3.2: A hard-fork example.

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.

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 first example

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 second example

In August 2010 an integer overflow bug was identified[5] in Bitcoin core v0.3.9 were billions of bitcoins could be erroneously sent. The community quickly coordinated and released v0.3.10 which fixed the issue by checking more thoroughly the integer limits.

The new version was soon run by the majority of nodes[6] and thus the new chain overtook the old erroneous one removing the inflation bug.

[5] https://bitcointalk.org/index.php?topic=822.0

[6] Back then all nodes were mining nodes.

Hard-fork third example

In March 2013, Bitcoin core v0.8 switched its 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.

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 and a merchant fell victim to a successful double spend but other than that the incident was painless.

3.5 Upgrading Bitcoin

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.

New proposals came up on how miners can signal agreement for particular upgrades. If there was enough consensus the upgrade would be activated. The first proposal was BIP-34[7] which allowed signalling for one upgrade decision at a time. It was later superseded by BIP-9[8] which allowed, among other benefits, more than one upgrade decision to be made. The current activation method is BIP-8[9] which is very similar to BIP-9 but more flexible.

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:

To add a new feature a block version number (e.g. 2) would be associated with it as well as a block height for activation.

[7] https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki
[8] https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki
[9] https://github.com/bitcoin/bips/blob/master/bip-0008.mediawiki

- If 750 out 1000 blocks[10] have block version of 2 then reject invalid v2 blocks (if no block height included)
- If 950 out of 1000 blocks[11] have block version of 2 then reject all block version 1 blocks

The BIPs activated with this signaling process were, BIP-32 (v2), BIP-65 (v3) and 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, potentially in parallel
- A structure is defined with:
  - name
  - bit, the block version bit used to signal for this change
  - starttime, time (Median Time-Past[12], BIP-113[13]) 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 66, 112 and 113 and proposal "segwit" that included BIPs 141, 143 and 147. There is a list[14] of all BIP-9 deployments (both past and current ones).

> The Bitcoin community opted to do only soft-fork upgrades to minimize potential disuption to the network. To that end extra precautions and effort is required in the upgrade design and implementation. A notable example is the *Segregated Witness* or *segwit* upgrade. The latter would be easier to implement as a hard-fork but since those are contentious they had to come up with a design that allowed the upgrade to happen as a soft-fork.

The segwit upgrade was quite contentious and there was a lot of political agendas and drama involved. A good walkthrough was published in Bitcoin Magazine[15] and a more detailed account of what happened is described in The Blocksize War book [**Bier2021-blocksize-wars**].

[10] 510 out of 1000 for testnet

[11] 750 out of 1000 for testnet

[12] Timestamp of the median of the past 11 blocks.

[13] https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki

[14] BIP-9 was used to activate proposal "csv" that contained BIPs 66, 112 and 113. There is a list of all BIP-9 deployments (both past and current ones).

[15] https://bitcoinmagazine.com/articles/long-road-segwit-how-bitcoins-biggest-protocol-upgrade-became-reality

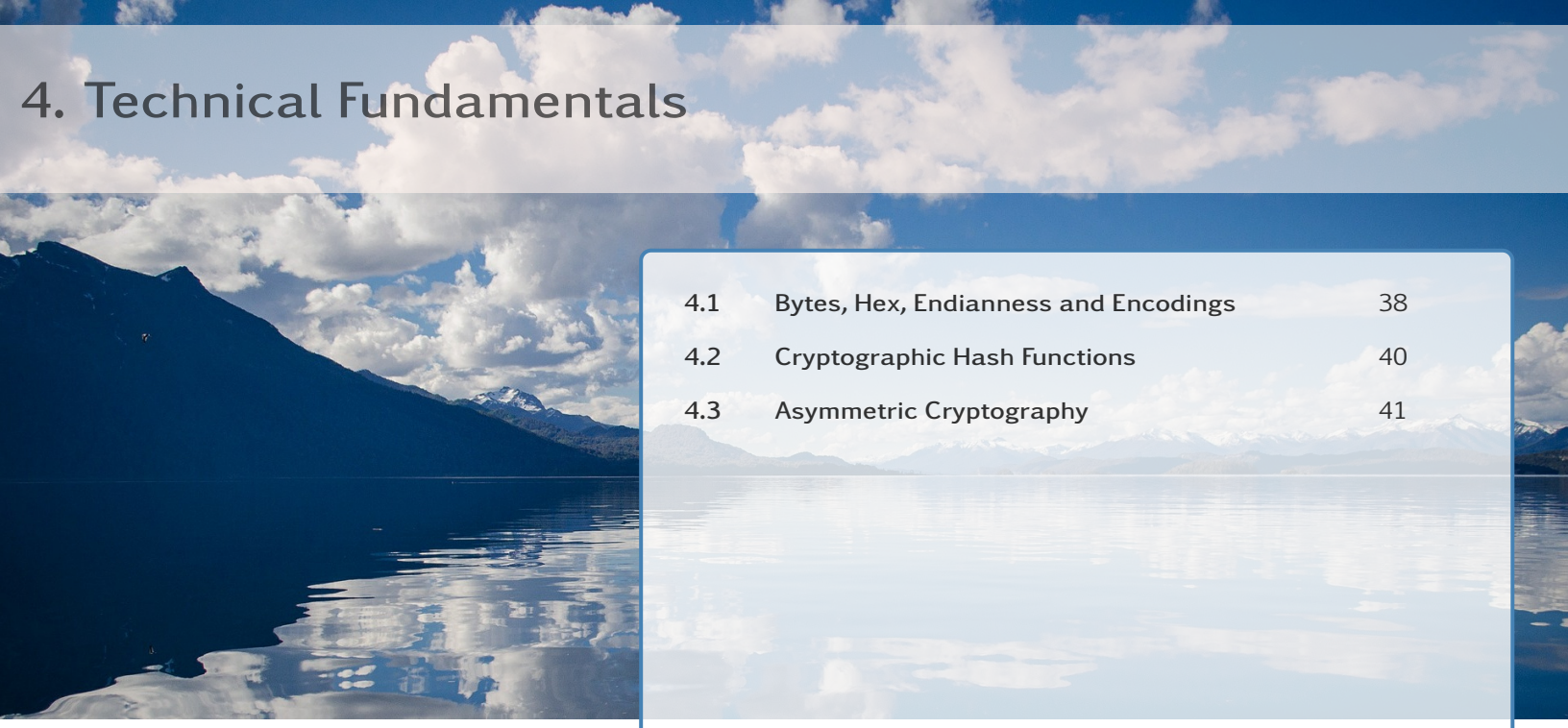
BIP-8

This is the current BIP used for upgrading. It has the following differences from BIP-9:

- Uses block height instead of timestamps for signalling; this makes it more predictable
- Threshold for activation is 90%
- Gives the option to reject or enforce the upgrade at the end of the timeout
  - **lockontimeout** or LOT=False (BIP-9 equivalent bahaviour)
  - LOT=True (enforces lock-in)

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

4. Technical Fundamentals



This chapter aims to provide some basic technical computer science background required for later material. It is of particular importance if you want to delve deeper and read the code of the Python **bitcoin-utils** *library*[1]*. It aims to concisely explain some fundamental concepts by providing examples.*

4.1 Bytes, Hex, Endianness and Encodings

Computers internally use the binary numeral system that consists of only two symbols: 0 and 1. A binary digit, or bit, is the basic unit of binary. Eventually, everything is represented in bits.

For ease of processing, bits are aggregated into bytes. Each byte consists of 8 bits. Thus, **01000001** represents a single byte. It is difficult (and much longer) to use/type binary, thus we typically use the hexadecimal numeral system or hex to represent bytes in a more human-friendly way. Hex has 16 possible symbols from 0 to F (0-9 and A-F). To represent 16 symbols we need exactly 4 bits (0 is 0000, 1 is 0001, . . . , F is 1111) and thus each byte can be represented by two hex digits. For example, **01000001** is equivalent to **41** in hexadecimal (0100 is hex number 4 and 0001 is hex number 1). You can experiment with conversions between binary, hexadecimal and decimal using various online tools[2].

```
>>> format(0, '04b')      # converts decimal 0 to binary with 0 padding for 4 bits
'0000'
>>> format(15, '04b')     # same for decimal 15
'1111'
>>> format(15, 'X')       # convert decimal 15 to hex string
'F'
>>> format(0x41, '08b')   # converts hex number 41 to binary with 0 padding for
                          # 8 bits
'01000001'
>>> 0x41                  # converts hex number 41 to decimal
65
```

[1] https://github.com/karask/python-bitcoin-utils

[2] https://www.rapidtables.com/convert/number/binary-to-hex.html

```
>>> 0x41 == '41'          # compares hex number 41 with string 41
False
>>> b'41' == '41'         # compares byte literal 41 with string 41
False
```

Listing 4.1: Python examples

One byte can represent up to $2^8 = 256$ numbers (0-255). Several bytes are needed to represent larger numbers. For example, decimal number 1000 is **1111101000** in binary which requires 10 bits. Computers operate at the byte level and thus 2 bytes (or 16 bits) will be required to represent this number; binary: **0000001111101000** and hex: **03E8**. Note that the byte ordering is important and it is called endianness[3]. The above example uses big-endian ordering, where the most significant byte comes first and the least significant byte comes last. This is the same way we order numbers in languages (in left-to-right scripts[4]). In little-endian the same number would be represented as **1110100000000011** and **E803**.

```
>>> import binascii
>>> format(0x03E8, '016b')           # convert hex number 03E8 to binary digits
                                     # with 0 padding for 16 bits
'0000001111101000'
>>> format(int('03E8', 16), '016b')  # convert hex string '03E8' to binary digits
                                     # as above
'0000001111101000'
>>> binascii.unhexlify('03E8')       # convert hex string 03E8 to binary
b'\x03\xe8'
>>> b'\x03\xe8'[::-1]                # reverse bytes to change endianness
b'\xe8\x03'
```

Listing 4.2: Python examples

> Internally Bitcoin uses little-endian byte order as it improves speed (most computers use little-endian byte ordering internally). Most hash function libraries (see next section) create hashes using big-endian and Bitcoin transmits those in that ordering. However, when hashes are displayed Bitcoin uses little-endian order! The latter might be because it treats them as integers to compare them faster.

As we already mentioned computers only know about binary. To display these numbers for human consumption we need to convert them into characters (i.e. text). In order to accomplish that we need character encodings that provide mappings between bit sequences and characters. Examples of such encodings are ASCII and UTF-8. UTF-8 is widely used nowadays and it provides an 8-bit mapping between (binary) numbers and characters. For example, character **A** is mapped to **01000001**. You can experiment with such encodings with online tools[5].

```
>>> import binasci
>>> 'A' == b'A'                      # A character is not a byte
```

[3] https://en.wikipedia.org/wiki/Endianness
[4] https://en.wikipedia.org/wiki/Writing_system#Directionality
[5] https://www.rapidtables.com/convert/number/ascii-to-binary.html

```
3  False
4  >>> b'41' == '41'.encode('utf-8')     # Unicode string literals are stored internally
5                                        # in binary for efficiency
6  True
7  >>> 'ε'.encode()                      # UTF-8 (the default) is a variable lenth
8                                        # encoding - 'ε' is stored as two bytes
9  b'\xce\xb5'
10 >>> '41'.encode()                     # converts UTF-8 string literal 41 to binary -
11                                       # '4' and '1' occupy 1 byte each
12 b'41'
13 >>> binascii.unhexlify('41')          # converts string hex value 41 to binary -
14                                       # the UTF-8 value for 'A' is 0x41
15 b'A'
16 >>> b'A' == b'\x41'                   # the bytes from 0x01 to 0x7f are confusingly
17                                       # specified with UTF-8 characters
18 True
19 >>> b'41' == b'\x34\x31'              # similarly for binary characters b'4' and
20                                       # b'1' or b'41'
21 True
22 >>> b'A'.decode('utf-8')              # convert binary to characters for displaying
23                                       # according to UTF-8 encoding
24 'A'
25 >>> binascii.hexlify(b'A')           # convert binary value to hex value (expressed
26                                       # in binary)
27 b'41'
28 >>> b'41'.decode()                    # decode to get as string (UTF-8 is the default)
29 '41'
30 >>> b'\xce\xb5'.decode()              # convert from bytes to UTF-8 characters (0xce
31                                       # 0xb5 maps to 'ε')
32 'ε'
```

Listing 4.3: Python examples

4.2 Cryptographic Hash Functions

A cryptographic hash function is a hash function that is suitable for cryptography. It is an one-way function that takes an arbitrary block of data and returns a fixed-size bit array, that is called the hash value or digest or digital fingerprint or just hash. It has the following properties:

- it is deterministic, i.e. the same block of data always returns the same hash
- it is quick to compute
- it is an one-way function; given the hash one cannot derive the original value unless they brute-force all possible values (which is close to impossible for large data sets)
- even a trivial change to the original data will change the resulting hash completely (it will appear uncorrelated to the previous hash)
- it is collision resistant; it is computational infeasible to find two different blocks of data with the same hash value

```
>>> import hashlib
>>> import binascii
>>> b1 = 'Bitcoin'.encode('utf-8')    # convert string to bytes according to UTF-8
                                      # encoding
>>> b1
b'Bitcoin'
>>> h1 = hashlib.sha256(b1).digest() # calculate the hash (or digest) of b1
>>> h1
b'\xb4\x05m\xf6i\x1f\x8d\xc7.V0-\xda\xd3E\xd6_\xea\xd3\xea\xd9)\x96\t\xa8&\xe24N' \
b'\xb6:\xa4'
>>> binascii.hexlify(h1).decode()     # converts bytes to hex (expressed in binary)
                                      # and then to string to display
'b4056df6691f8dc72e56302ddad345d65fead3ead9299609a826e2344eb63aa4'
>>> b2 = 'bitcoin'.encode('utf-8')    # convert string to bytes according to UTF-8
                                      #encoding
>>> b2
b'bitcoin'
>>> h2 = hashlib.sha256(b2).digest() # calculate the hash (or digest) of b2
>>> h2
b'k\x88\xc0\x87$z\xa2\xf0~\xe1\xc5\x95k\x8e\x1a\x9fL\x7f\x89*p\xe3$\xf1\xbb=\x16' \
b'\x1e\x05\xca\x10{'
>>> binascii.hexlify(h2).decode()     # converts bytes to hex (expressed in binary)
                                      # and then to string to display
'6b88c087247aa2f07ee1c5956b8e1a9f4c7f892a70e324f1bb3d161e05ca107b'
```

Listing 4.4: Python examples

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.

> Bitcoin is using two hashing functions: SHA-256 and RIPEMD-160 which create a hash value of 256 and 160 bits respectively (or 32 and 20 bytes or 64 and 40 hex characters).

4.3 Asymmetric Cryptography

Asymmetric cryptography or public key cryptography[6] 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.

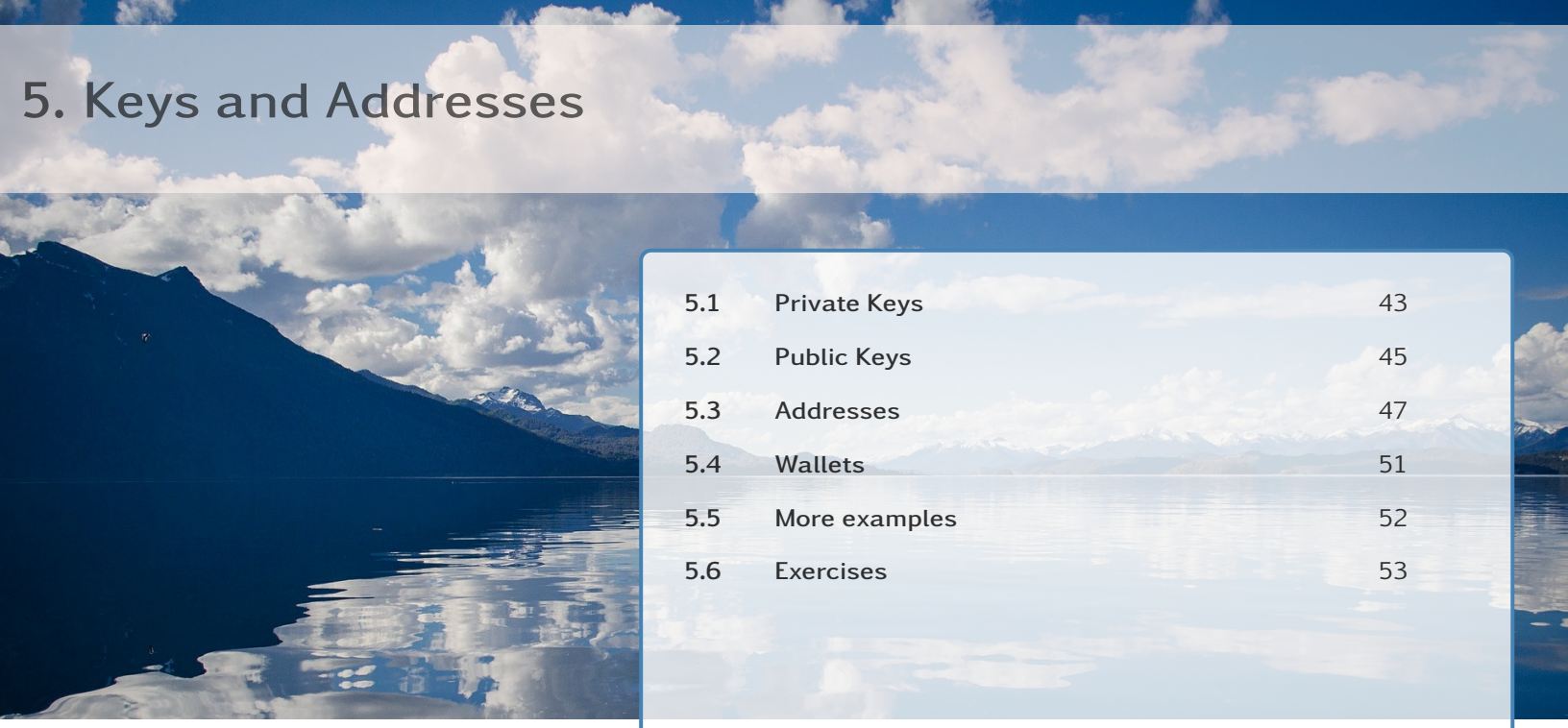
[6] https://en.wikipedia.org/wiki/Public-key_cryptography

- Integrity: While anyone can view the contents of a signed message no one can modify it since the signature will be invalidated.
- Non-Repudiation: Signing or encrypting a message cannot be refuted by the author once the message has been sent, assuming the private key is secure.

> Bitcoin does not use encryption 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. Integrity and non-repudiation also apply to transaction signing.

There are several different algorithms for asymmetric cryptography, like RSA and ECDSA. ECDSA, which Bitcoin uses, has the property that a private key can be used to calculate the corresponding public key.

5. Keys and Addresses



In this chapter we introduce Bitcoin's keys and addresses and describe how they are created and the rationale behind that process. We then go through different wallet types and how these keys can be used in practice.

5.1 Private Keys

Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA)[1] to create its private-public key pairs. The exact elliptic curve parameters used in Bitcoin are defined by secp256k1[2].

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

The ECDSA private key in Bitcoin is just a very large 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.

To display a private key (the bytes) we need to format it appropriately. It could be displayed in hex but the most common format used to display a private key is Wallet Import Format (WIF) or WIF-compressed (WIFC); both are a Base58Check[3] encoding of the ECDSA key; Base58[4] with a 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[5].

[1] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
[2] https://en.bitcoin.it/wiki/Secp256k1
[3] https://en.bitcoin.it/wiki/Base58Check_encoding
[4] https://en.wikipedia.org/wiki/Base58
[5] Note that the segregated witness upgrade allows only compressed public keys.

The WIFC will be 33 bytes long. The compression is happening when creating the public key which will be 33 bytes instead of 65 bytes.

The following is pseudocode of the process that converts the private key to WIF.

```
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 )
```

Note that all the above functions operate on big-endian bytes. The network prefix specifies the Bitcoin network that this private key would be used[6]. The Base58 WIF prefix depends on the network prefix and whether it is compressed or not, as shown in table 5.1.

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

Table 5.1: Private keys network prefixes

As an example let us use the following hexadecimal number[7]:

```
0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b
```

You can now consult table 5.2 for its compressed version and the corresponding WIF and WIF-C.

HEX	0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b
HEX-C	0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b**01**
WIF	91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p
WIF-C	cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo

Table 5.2: Private key network prefix examples

[6]The same private key can be of course used in both mainnet and testnet or even other compatible cryptocurrencies by using the appropriate prefix when generatig the WIF.

[7]In decimal: 6273083586486421860511804118443655246000698298582245067579657211. It is important to understand that a random number with enough entropy is required for your private key to be secure. A number representing your date of birth or your name's characters, etc. will be found immediately by software and you will lose your funds.

Let's use the bitcoin-utils library to construct the WIF and WIFC.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import PrivateKey
>>> setup('testnet')                                    # use testnet parameters
'testnet'
>>> secret_exponent = 0x0dde70823a4bb0ca3bd75a2010e8d5dc091185e73d8b4257a981c695a3eba95b
>>> priv = PrivateKey(secret_exponent = secret_exponent)
>>> priv.to_wif(compressed=False)
'91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p'
>>> priv.to_wif(compressed=True)                        # the default
'cN3fHnPVw4h7ZQSRz2HgE3ko69LTaZa5y3JWpFhoXtAke4MiqVQo'
```

Listing 5.1: Example of creating WIF and WIFC using Python

The actual Python implementation of the functionality demonstrated above can be found at **to_wif()**[8] on github. You can also check how we can get to the private key bytes from WIF in **_from_wif()**[9]. Feel free to consult the rest of the code and/or the examples in the repository.

Another tool that you can use from the command line is BX[10]. It has extensive capabilities including creating WIFs.

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

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

5.2 Public Keys

In ECDSA a public key is generated from the private key. Elliptic curves operate over finite fields[11] and thus all points on the curve are limited to integer coordinates; a finite field is typically accomplished by applying modulo p, where p is a prime number. The specific curve that Bitcoin uses (secp256k1) is $y^2 = x^3 + 7$ (see figure 5.1[12]). Then the public key P is generated by multiplying, using elliptic curve multiplication, the private key k with a special constant G called the generator point[13]: $P = k * G$. Elliptic curve multiplication of an integer with a point results in another point in the curve, which is the public key.

[8] https://github.com/karask/python-bitcoin-utils/blob/42875a3fa90d267f2e5e17e017cb28fc8a90c5a8/bitcoinutils/keys.py\#L169-L193

[9] https://github.com/karask/python-bitcoin-utils/blob/42875a3fa90d267f2e5e17e017cb28fc8a90c5a8/bitcoinutils/keys.py\#L129-L166

[10] https://github.com/libbitcoin/libbitcoin-explorer/wiki/Download-BX

[11] https://en.wikipedia.org/wiki/Finite_field

[12] From https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc

[13] This is a special point in the elliptic curve that is pre-defined in secp256k1.

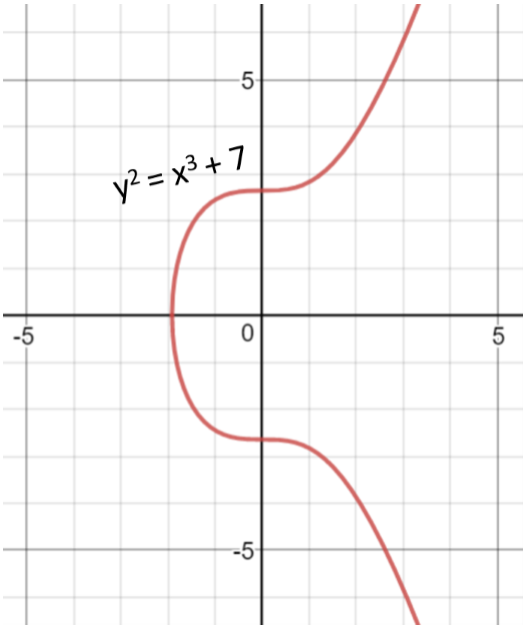


Figure 5.1: The secp256k1 ECDSA curve.

The public key is a point P in the elliptic curve. $P = (x, y)$, where both x and y are 32-byte integers. Thus a public key can be expressed with 64 bytes. In Bitcoin, we encode a public key by a prefix that specifies some extra information.

> Remember that we can represent a public key in two forms, compressed and uncompressed. This is where we can reduce the size of the blockchain by using the compressed form.

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.

Since the curve is mirrored in the x axis the y coordinate can only take 2 values for a specific x (positive/negative). Thus, an encoded compressed public key is only 33 bytes long and has only the x coordinate with a prefix of **0x02** (when y is positive/even) or **0x03** (when y is negative/odd).

Let's use the bitcoin-utils library to construct a private key object from a WIF and use that to create a public key object to show its two forms.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import PrivateKey, PublicKey
>>> setup('testnet')
'testnet'
>>> priv = PrivateKey.from_wif('91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p')
>>> pub = priv.get_public_key()
>>> pub.to_hex()                # default is compressed form
'02c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7'
>>> pub.to_hex(compressed=False)
'04c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7\\
```

```
11  addc4cbba6656a4be4bc6933a6af712b897a543a09c4b899e5f7b943d38108a8'
```

Listing 5.2: Python example to generate compressed and uncompressed public keys

You can see in lines 10-11 in listing 5.2 that the uncompressed public key has the same x coordinate as the compressed one plus the y coordinate.

To create the PublicKey from the PrivateKey object we make use[14] of the Python ECDSA library as can be seen in **get_public_key()** on github[15]. The PublicKey object holds the x and y coordinates and can convert accordingly. It checks if y is even or odd and prefixes it with $0x02$ and $0x03$ respectively. You can check the code of **to_hex()** on github[16].

We can get the public keys from the private keys using BX again.

```
$ ./bx wif-to-public 91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p
04c1acdac799fb0308b4b6475ddf7967676759d31484ab55555482472f3bc7c3e7\\
addc4cbba6656a4be4bc6933a6af712b897a543a09c4b899e5f7b943d38108a8

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

5.3 Addresses

Addresses can be shared to anyone who wants to sent 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[17].

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 a future chapter.

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

- shorter addresses
- quantum computer resistance

> Until one spends from an address the public key will never appear on the blockchain and thus to potential attackers 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. Note, however, that even if that is the case the majority of addresses would be hacked thus destroying trust in (and the value of) the network anyway!

[14] It is always recommended to reuse well-tested cryptography libraries than implementing your own.

[15] https://github.com/karask/python-bitcoin-utils/blob/fb0849f81117943563b17f1870a9607d48ca9653/bitcoinutils/keys.py\#L351-L355

[16] https://github.com/karask/python-bitcoin-utils/blob/fb0849f81117943563b17f1870a9607d48ca9653/bitcoinutils/keys.py\#L453-L469

[17] Or for segwit addresses bc1 and tb1 for mainnet and testnet respectively.

Legacy Addresses

An address is really just the hash of the public key, called public key hash (or PKH). That is how it is represented on the blockchain. The way we format addresses to display them (starting with 1, m/n, etc.) are just for our convenience. The format that we use is the Base58Check encoding of the public key hash; Base58 with version prefix to specify the network and a 32-bit checksum.

The following is pseudocode of the process that converts the public key to public key hash and then address:

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

Note that all the above functions operate on big-endian bytes. The network prefix specifies the Bitcoin network that this private key would be used. The Base58 WIF prefix depends on the network prefix and whether it is compressed or not, as shown in table 5.3.

	Mainnet		Testnet	
	Network Prefix	**Base58 Prefix**	**Network Prefix**	**Base58 Prefix**
P2PKH	0 | 0x00	1	111 | 0x6f	m or n
P2SH	5 | 0x05	3	196 | 0xc4	2

Table 5.3: Addresses network prefixes

P2PKH stands for Pay to Public Key Hash and is the typical legacy address. **P2SH** stands for Pay to Script Hash and is the typical legacy script hash address; an address that is protected with an arbitrary script rather than just a private-public key pair as in the P2PKH case.

Let's use the bitcoin-utils library to construct some addresses; compressed and uncompressed.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import PrivateKey, PublicKey, P2pkhAddress
>>> setup('testnet')
'testnet'
>>> priv = PrivateKey.from_wif('91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p')
>>> pub = priv.get_public_key()
>>> addr1 = pub.get_address()
>>> addr2 = pub.get_address(compressed=False)
>>> addr1.to_string()
'n42m3hGC52QTChUbXq3QAPVU6nWkG9xuWj'
>>> addr2.to_string()
'n2JjAgC6UqFf8DvsZXhWcyNzm8w8YKj7MQ'
```

Listing 5.3: Python example to generate compressed and uncompressed addresses

> Since the hash of a compressed public key would be different from the hash of an uncompressed public key we have two distinct legacy addresses.

The actual Python implepmentation of converting a public key hash (the application of SHA256 and then RIPEMD160, also called HASH160) can be found at **_to_hash160()** on github[18]. For code to convert from public key hash to address consult **to_string()**[19]. Feel free to consult the rest of the code and/or the examples in the repository for segwit address creation, etc.

Native Segregated Witness 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 section 6.5.

Native segwit addresses use a different format to display the public key hash called Bech32 encoding[20] (instead of Base58check). For code implementation look at the Python reference implementation[21] by Pieter Wuille.

The network prefix specifies the Bitcoin network that this address would be used. Specifically, mainnet addresses start with **bc1**, testnet with **tb1** and regtest[22] with **bcrt1**. The size of the address differentiates between public key hashes and script hashes; P2WPKH are 20 bytes long and P2WSH are 32 bytes long.

Let's use the bitcoin-utils library to construct a segwit address.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import PrivateKey, PublicKey, P2wpkhAddress
>>> setup('testnet')
'testnet'
>>> priv = PrivateKey.from_wif('91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p')
>>> pub = priv.get_public_key()
>>> addr = pub.get_segwit_address()
>>> addr.to_string()
'tb1q7m6ak6k050sxzxjjekhey73k0f3rqnxsqa08k2'
```

Listing 5.4: Example of displaying a segwit address from a public key using Python

Nested Segregated Witness Addresses

When segwit was introduced a lot of wallets did not support the new bech32 addresses, so users could not use them to send funds to segwit addresses. To remedy that, nested segwit addresses could be used.

[18] https://github.com/karask/python-bitcoin-utils/blob/52b7d906f2db8ec4ed4945a04b7e4da2d1db369c/bitcoinutils/keys.py\#L589-L597

[19] https://github.com/karask/python-bitcoin-utils/blob/52b7d906f2db8ec4ed4945a04b7e4da2d1db369c/bitcoinutils/keys.py\#L802-L824

[20] https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki

[21] https://github.com/karask/python-bitcoin-utils/blob/52b7d906f2db8ec4ed4945a04b7e4da2d1db369c/bitcoinutils/bech32.py

[22] Legacy addresses use the same prefix as testnet.

Effectively you could nest or wrap a segwit address into a P2SH address. As already mentioned P2SH addresses can be created from arbitrary scripts and thus could also include a witness script. Again, P2SH and segwit are outside our scope for now and will be explained in more detail in section 6.5.

> After the segwit upgrade one needs to choose the type of address to create (e.g. when using **getnewaddress**). The supported types where **legacy**, **p2sh-segwit** and **bech32**. The default, starting from version 0.16.0, was nested segwit addresses (**p2sh-segwit**) and from version 0.19 onwards the default is segwit addresses (**bech32**). The default can be changed in the configuration using **addresstype**.

Vanity Addresses

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

```
1  import random
2  from bitcoinutils.setup import setup
3  from bitcoinutils.keys import PrivateKey
4
5  setup('mainnet')
6
7  vanity_string = '1KK'
8  found = False
9  attempts = 0
10
11 while(not found):
12     p = PrivateKey(secret_exponent = random.getrandbits(256))
13     a = p.get_public_key().get_address()
14     print('.', end='', flush=True)
15     attempts += 1
16     if(a.to_string().startswith(vanity_string)):
17         found = True
18
19 print("\nAttempts: {}".format(attempts))
20 print("Address: {}".format(a.to_string()))
21 print("Secret Key: {}".format(p.to_wif()))
```

Listing 5.5: Example of creating a vanity address using Python

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 additional 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 a vanity address that starts with **1Kostas** 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 arithmetics 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. Figure 5.2 illustrates the process to securely get to the private key that corresponds to the vanity address.

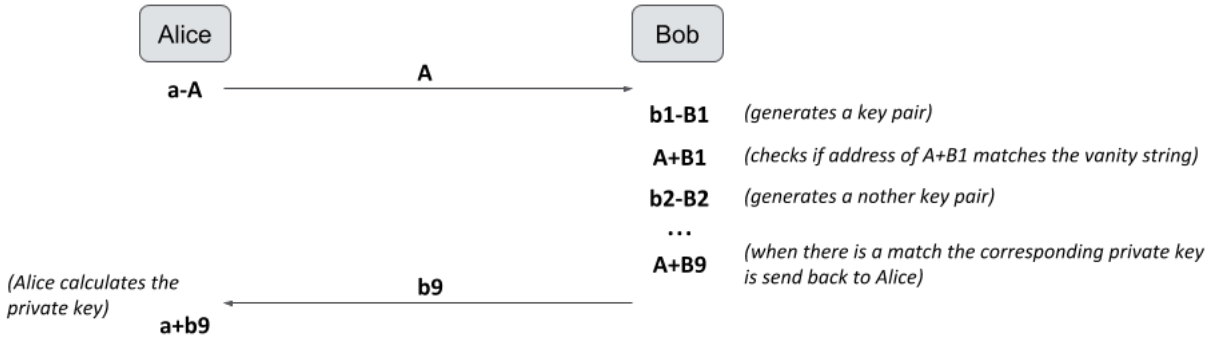


Figure 5.2: Example of how a vanity address pool securely shares the private key to a user.

- First Alice creates a key pair and sends the public key **A** to Bob.
- Bob creates a key pair and adds the new public key **B1** to Alice's key. Then uses the resulting public key to generate an address. If this address does not match the vanity string required by Alice, it repeats the process by creating another key pair, and so on. When a match is found the respective private key, e.g. **b9**, is send to Alice.
- Now Alice, and only Alice, can construct a private key that corresponds to the vanity address by adding it to her private key.

5.4 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:

Non-deterministic 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.

Nowadays, most wallets are *Hierarchical Deterministic (HD)* since they offer more flexibility, easier backups and enchanced security in certain use cases. HD wallets are described in detail in *Bitcoin Improvement Proposals (BIPs)* 32[23], 43[24] and 44[25].

5.5 More examples

This section will provide more examples of how to create and use keys and addresses using the **bitcoin-utils** Python library.

Create a P2SH-P2WPKH address

In this example we will create and display a P2SH address that wraps a P2WPKH script. For more details on what P2SH and P2WPKH are please refer to sections 6.4 and 6.5 respectively.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import PrivateKey, P2shAddress
>>> setup('testnet')
'testnet'
>>> native_p2wpkh = PrivateKey.from_wif('cTmyBsxMQ3vyh4J3jCKYn2A'\
...         'u7AhTKvqeYuxxkinsg6Rz3BBPrYKK').get_public_key().get_segwit_address()
>>> print("P2WPKH:", native_p2wpkh.to_string())
P2WPKH: tb1qsd4ax84vhem5hxgxus2u232nw9ylgftkz0szf2
>>> nested_p2wpkh = P2shAddress.from_script(native_p2wpkh.to_script_pub_key())
>>> print("P2SH(P2WPKH):", nested_p2wpkh.to_string())
P2SH(P2WPKH): 2N8Z5t3GyPW1hSAEJZqQ1GUkZ9ofoGhgKPf
```

Sign message with private key and verify

We can use a private key to sign a message. In asymmetric cryptography, the recipient can then use the corresponding public key to verify that the message was not tampered with.

```
>>> from bitcoinutils.setup import setup
>>> from bitcoinutils.keys import P2pkhAddress, PrivateKey, PublicKey
>>> setup('testnet')
'testnet'
>>> priv = PrivateKey.from_wif('91h2ReUJRwJhTNd828zhc8RRVMU4krX9q3LNi4nVfiVwkMPfA9p')
```

[23] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
[24] https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki
[25] https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

```
>>> address = priv.get_public_key().get_address()
>>> message = "The test!"
>>> signature = priv.sign_message(message)
>>> print("The signature is:", signature)
The signature is: INzSwXNYNUkPFIms1SFDzvqoib3ZdODcaBSZHvx5e4z1wc64cF1dVMZbDFtZxYB1D\\
L/dsjK2iBD5qAf7VcmSdQo=
>>> if PublicKey.verify_message(address.to_string(), signature, message):
...     print("The signature is valid!")
... else:
...     print("The signature is NOT valid!")
...
The signature is valid!
```

Listing 5.6: Use public key to sign a message and them verify

> We verify using the address instead of the public key. In ECDSA cryptography the public key can be reconstructed given the signature and the public key hash (or address).

5.6 Exercises

Exercise 5.1 Describe possible outcomes of mistyping a Bitcoin address when trying to send some bitcoins.

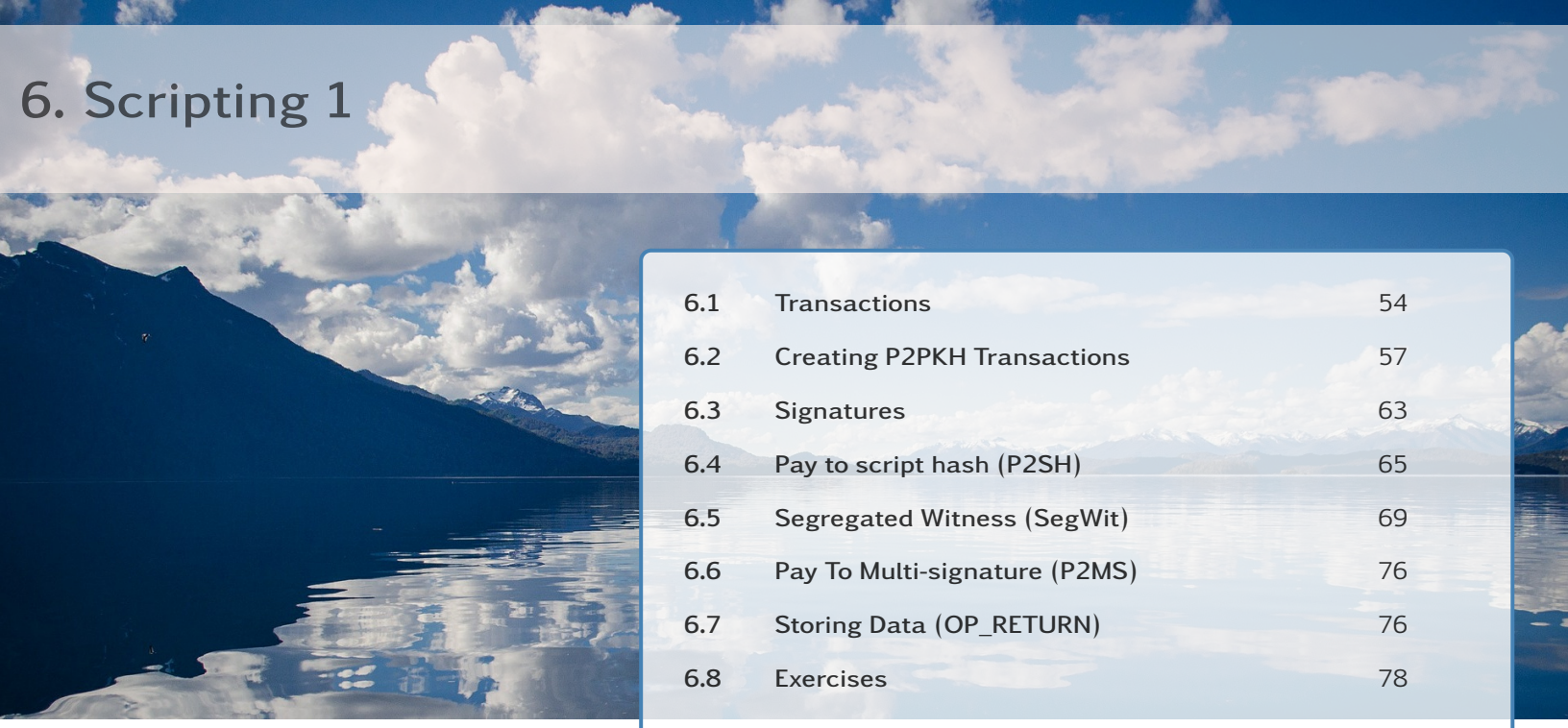
Exercise 5.2 Use a *vanity generator*, like https://github.com/samr7/vanitygen, to create some addresses.

Exercise 5.3 Use Python and the **bitcoin-utils** library to create a simple vanity generator.

Exercise 5.4 Write a function that creates a Bitcoin address given a public key. The only Python libraries that you are allowed to use are **hashlib** for the hashing algorithms and **binascii** to convert between hexadecimal and bytes.

Exercise 5.5 Create and display a P2SH address than contain a P2PK script using any random private key.

6. Scripting 1



This chapter goes deeper into what constitutes a transaction and how scripting is used to lock bitcoins and later unlock them to spend them. Several examples are provided on how to create transactions by calling a node's API or programmatically.

6.1 Transactions

A transaction sends bitcoins from one address to another and it consists of 1+ inputs and 1+ outputs. The inputs of a transaction consist of outputs of previous transactions. When an output is spend it can never be used again[1]. All the bitcoins are transferred elsewhere (to a recipient, back to yourself as change, etc.). Outputs that are available to be spend are called *Unspent Transaction Outputs (UTXOs)* and Bitcoin nodes keep track of the complete UTXO set.

> Each time funds are sent to an address a new output (UTXO) is created. Thus, the balance of an address depends on all the UTXOs that correspond to it. Bitcoin wallets hide UTXOs to make the whole experience friendlier but some wallets allow you to specify which UTXOs you want to spend if needed. When we create transactions programmatically we will deal primarily with UTXOs.

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

The spending 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.

[1] Think of cash. If you give a 20 euro note you can never reuse it. You might be given change but it will be different notes or coins.

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**.

The standard transaction output types supported by the Bitcoin protocol are:

- P2PK (Pay to Public Key - not used anymore)
- P2MS (legacy multisignature transactions; now P2SH/P2WSH/P2TR are used instead)
- P2PKH (Pay to Public Key Hash)
- P2SH (Pay to Script Hash)
- P2WPKH (Pay to Witness Public Key Hash)
- P2WSH (Pay to Witness Script Hash)
- P2TR (Pay to Taproot)
- OP_RETURN (allows for storing up to 80 bytes in an output)
- Non-standard[2] (any other valid transaction)

The most common transaction output type offering a standard way of transferring bitcoins around is P2PKH (and P2WPKH), 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 P2PK but that is not used anymore. Another very important transaction output type is P2SH (and P2WSH) which allows locking scripts of arbitrary complexity to be used.

To define a locking and unlocking script we make use of a scripting language, simply called *Script*[3]. This relatively simple language consists of several operations each of them identified by an *opcode* in hexadecimal. It is a simple stack-based language that uses reverse polish notation (e.g. `2 3 +`) and does not contain potentially dangerous programming constructs, like loops; it is a domain-specific language.

P2PKH

Let's examine the standard transaction of spending a Pay to Public Key Hash. The locking script (**scriptPubKey**) that secures the funds in a P2PKH address is the following:

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

As we have seen in section 5.3 the public key hash (PKHash) can be derived from the Bitcoin address and vice versa. Thus, the above script locks the funds that have been sent in the address that corresponds to that PKHash.

To spend the funds the owner of the private key that corresponds to that address/PKHash needs to provide an unlocking script that if we prepend to the locking script the whole script will evaluate to true. An unlocking script for a P2PKH will look like this:

[2]Valid non-standard transactions (containing scripts other than those defined by the standard transaction output type scripts) are rejected and not relayed by nodes. However, they can be mined if it is arranged with a miner.

[3]https://en.bitcoin.it/wiki/Script

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

Using the private key we provide an ECDSA signature of part of the transaction that we create (see section 6.3 for more details). We also provide the public key[4] for additional verification.

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.

Validation of P2PKH Spending

The validation process is described below in a step by step explanation during script execution. In each step, the script element evaluated will be highlighted (left column) and the current stack (right column) will also be displayed.

Step 0: Execution starts. Stack is empty.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	
Step 1: First element is evaluated. It consists of data so it goes into the stack.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature>
Step 2: Second element is also data and goes into the stack.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature> <PublicKey>
Step 3: Next element is an operator that duplicates the top element of the stack.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature> <PublicKey> <PublicKey>
Step 4: Next element is an operator that calcuates the HASH160 of the top stack element. HASH160 is equivalent to RIPEMD160(SHA256(element)) which is what is needed to calculate the PKH from a public key.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature> <PublicKey> <PKHash>
Step 5: Next element is data and it is pushed into the stack.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature> <PublicKey> <PKHash> <PKHash>
Step 6: Next element is an operator that checks if the top two elements of the stack are equal and fails the script if they are not. Effectively this validates that the public key provided is indeed the one that corresponds to the PKH (or address) that we are trying to spend.	

[4] As we have already discussed in section 5.3 the public key only appears in the blockchain after we spend from an address. This is where it appears!

<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	<Signature> <PublicKey>
Step 7: Next element is an operator that expects two elements from the stack; a signature and a public key that corresponds to that signature. If the signature is valid it returns true, otherwise false.	
<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG	OP_TRUE

Since the script finished and the only element in the stack is now **OP_TRUE**[5] the node validated ownership of the UTXO and it is allowed to be spent. Success!

To help clarify how addresses, locking scripts and UTXOs relate please see figure 6.1. 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.

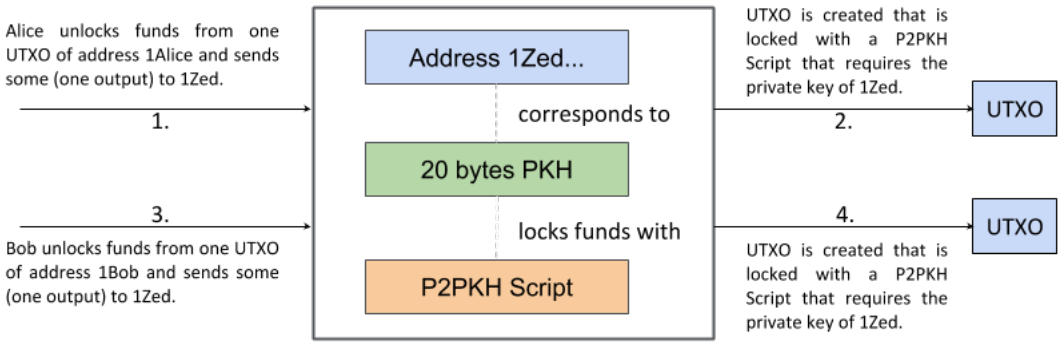


Figure 6.1: UTXOs, PKHashes and Addresses relationships.

This section explained how funds residing in UTXOs are locked/unlocked and how scripts are evaluated for validation. In the next section we will go through several examples of how we can create simple transactions programmatically.

6.2 Creating P2PKH Transactions

In the previous section we went through transactions, their inputs and outputs and how the funds are locked. In this post we will go through different ways of creating a simple payment transaction from the command line and then programmatically.

[5]Or OP_1, i.e. true. All the operators, or opcodes, and their explanations can be found at https://en.bitcoin.it/wiki/Script

Automatically Create a Transaction

We can use Bitcoin's build-in command **sendtoaddress** to send bitcoins to an address.

```
$ ./bitcoin-cli sendtoaddress mnB6gSoVfUAPu6MhKkAfgsjPfBWmEEmFr3 0.1
18f23a2c3bea97d30e0e09376222b6888943e7dc86df43ff5dfa1ff59c10d8ec
```

In this example we use the node to send **0.1** bitcoins to a testnet address. Notice that we do not specify any details on which UTXOs to spend from. The node wallet will decide which UTXOs it will spend and in which address the change (there is almost always change) will go; i.e. we do not have any control when sending funds this way.

> Notice that the result is the transaction identifier (txid) of this transaction.

Creating a Transaction using a Node

In this example we want to select the inputs explicitly. We need to know the txids and the output indexes (vout). As an example, we can get those with:

```
$ ./bitcoin-cli listunspent 0
[
  {
    "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
    "vout": 0,
    "address": "n1jnMQCyt9DHR3BYKzdbmXWM8M5UvH9nMW",
    "account": "",
    "scriptPubKey": "76a914ddcf9faf5625d6a96790710bbcef98af9a8719e388ac",
    "amount": 1.30000000,
    "confirmations": 0,
    "spendable": true,
    "solvable": true
  }
  ...
]
```

The above command lists all UTXOs (even those with 0 confirmations; i.e. in the mempool). Now we can create a transaction specifying the UTXO that we want to spend.

```
$ ./bitcoin-cli createrawtransaction '''
> [
>  {
>   "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
>   "vout": 0
>  }
> ]
```

```
> ''' '''
> {
>  "mqazutWCSnuYqEpLBznke2ooGimyCtwCh8": 0.2
> }'''
0100000001403918...efbe09488ac00000000
```

The result is the serialized raw transaction in hexadecimal. Note that this is not signed yet. To see the details of the raw transaction we can use:

```
$ ./bitcoin-cli decoderawtransaction 0100000001403918...efbe09488ac00000000
{
  "txid": "a7b54334096108e8f69ecfa19263cfbf2f12210165ef5fc2e98ef8e4e466392e",
  "hash": "a7b54334096108e8f69ecfa19263cfbf2f12210165ef5fc2e98ef8e4e466392e",
  "size": 85,
  "vsize": 85,
  "version": 1,
  "locktime": 0,
  "vin": [
   {
    "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
    "vout": 0,
    "scriptSig": {
      "asm": "",
      "hex": ""
    },
    "sequence": 4294967295
   }
  ],
  "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"
      ]
    }
   }
  ]
}
```

We can confirm that this is unsigned because the unlocking script or **scriptSig** is empty. We now need to sign this transaction before it becames a valid transaction.

```
$ ./bitcoin-cli signrawtransactionwithwallet 01000000014039...be09488ac00000000
{
  "hex": "0100000001403918270...38aefbe09488ac00000000",
  "complete": true
}
```

Now we have the final signed raw transaction (in attribute **hex**). If we use **decoderawtransaction** now you will see the unlocking script is properly set. We can test if this is a valid transaction before sending it to the node for broadcasting with the **mempoolaccept** command. Finally, we need to send it to the node for broadcasting.

```
$ ./bitcoin-cli sendrawtransaction 0100000001403918270...38aefbe09488ac00000000
error code: -26, error message:, 256: absurdly-high-fee
```

In this instance we get an error saying that the transaction has an exceptionally high fee. We have not specified any output for change so **1.1** bitcoins would be given to miners (**1.3-0.2**). Most wallets have similar protection mechanisms to help safeguard from user errors.

Using HTTP JSON-RPC

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 the **rpcallowip** configuration option.

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

Then we could use a tool like **curl** to make the JSON-RPC request:

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

{
  "result": 1746817,
  "error": null,
  "id": "curltest"
}
```

Thus, we can also send the commands seen before to construct transactions via JSON-RPC.

Calling Node Commands Programmatically

Library **python-bitcoin-utils**[6] wraps a Bitcon RPC library[7] and provides a proxy object that allows a Python program to call all the CLI commands programmatically via JSON-RPC. For example:

```
from bitcoinutils.setup import setup
from bitcoinutils.proxy import NodeProxy

# always remember to setup the network
setup('testnet')

# get a node proxy using default host and port
proxy = NodeProxy('rpcuser', 'rpcpw').get_proxy()

# call the node's getblockcount JSON-RPC method
count = proxy.getblockcount()

print(count)
```

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

Creating Transactions Programmatically

The Bitcoin node allows the creation of basic transactions but it does not support arbitrary scripts. We can create those programmatically by explicitly specifying the locking/unlocking conditions. We will use the **python-bitcoin-utils** library to demonstrate.

There are several examples included in the library that you can consult, like the most common transaction, a P2PKH payment[8] with one input and two outputs (the second output being the change).

```
# Copyright (C) 2018-2020 The python-bitcoin-utils developers
#
# This file is part of python-bitcoin-utils
#
# It is subject to the license terms in the LICENSE file found in the top-level
# directory of this distribution.
#
# No part of python-bitcoin-utils, including this file, may be copied,
# modified, propagated, or distributed except according to the terms contained
# in the LICENSE file.

from bitcoinutils.setup import setup
from bitcoinutils.utils import to_satoshis
from bitcoinutils.transactions import Transaction, TxInput, TxOutput
```

[6] The library can be installed directly with **pip install bitcoin-utils** in your working Python environment

[7] https://github.com/jgarzik/python-bitcoinrpc

[8] https://github.com/karask/python-bitcoin-utils/blob/b31c82e7005e06db7f780688cfcd9332d479f39d/examples/p2pkh_transaction.py

```
15 from bitcoinutils.keys import P2pkhAddress, PrivateKey
16 from bitcoinutils.script import Script
17
18 def main():
19     # always remember to setup the network
20     setup('testnet')
21
22     # create transaction input from tx id of UTXO (contained 0.4 tBTC)
23     txin = TxInput('fb48f4e23bf6ddf606714141ac78c3e921c8c0bebeb7c8abb2c799e9ff96ce6c', 0)
24
25     # create transaction output using P2PKH scriptPubKey (locking script)
26     addr = P2pkhAddress('n4bkvTyU1dVdzsrhWBqBw8fEMbHjJvtmJR')
27     txout = TxOutput(to_satoshis(0.1), Script(['OP_DUP', 'OP_HASH160', addr.to_hash160(),
28                                   'OP_EQUALVERIFY', 'OP_CHECKSIG']) )
29
30     # create another output to get the change - remaining 0.01 is tx fees
31     # note that this time we used to_script_pub_key() to create the P2PKH
32     # script
33     change_addr = P2pkhAddress('mmYNBho9BWQB2dSniP1NJvnPoj5EVWw89w')
34     change_txout = TxOutput(to_satoshis(0.29), change_addr.to_script_pub_key())
35     #change_txout = TxOutput(to_satoshis(0.29), Script(['OP_DUP', 'OP_HASH160',
36     #                                    change_addr.to_hash160(),
37     #                                    'OP_EQUALVERIFY', 'OP_CHECKSIG']))
38
39     # create transaction from inputs/outputs -- default locktime is used
40     tx = Transaction([txin], [txout, change_txout])
41
42     # print raw transaction
43     print("\nRaw unsigned transaction:\n" + tx.serialize())
44
45     # use the private key corresponding to the address that contains the
46     # UTXO we are trying to spend to sign the input
47     sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')
48
49     # note that we pass the scriptPubkey as one of the inputs of sign_input
50     # because it is used to replace the scriptSig of the UTXO we are trying to
51     # spend when creating the transaction digest
52     from_addr = P2pkhAddress('myPAE9HwPeKHh8FjKwBNBaHnemApo3dw6e')
53     sig = sk.sign_input( tx, 0, Script(['OP_DUP', 'OP_HASH160',
54                                        from_addr.to_hash160(), 'OP_EQUALVERIFY',
55                                        'OP_CHECKSIG']) )
56
57     # get public key as hex
58     pk = sk.get_public_key().to_hex()
59
60     # set the scriptSig (unlocking script)
61     txin.script_sig = Script([sig, pk])
62     signed_tx = tx.serialize()
63
64     # print raw signed transaction ready to be broadcasted
65     print("\nRaw signed transaction:\n" + signed_tx)
66
67
68 if __name__ == "__main__":
69     main()
```

This produces the serialized raw transaction in hexadecimal. We can then use **sendrawtransaction** to broadcast this to the Bitcoin network. Notice that the transaction identifier

used to create the input is hardcoded for simplicity; in an actual program we would call **listunspent** to find the available UTXOs.

6.3 Signatures

In this section we will explain how and what needs to be signed to prove ownership of funds residing in specific addresses.

When we create a new transaction we need to provide a signature for each UTXO that we want to spent. For a P2PKH UTXO 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. There are different ways to sign inputs of a transaction so as to provide different commitments. For example: "I agree to spend this input and sign it as long as no one can change the outputs I specified". To specify these commitments, i.e. which parts of the transaction will be signed, we add a special 1 byte flag called SIGHASH at the end of the DER signature.

Each transaction input needs to be signed separately from others. Parts of the new transaction will be hashed to create a digest and the digest is what is signed and included in the unlocking script. To determine which parts are hashed the following rules are followed:

- all other inputs' unlocking scripts (scriptSigs) should be empty
- the input's unlocking script (scriptSig), the one that we sign, should be set to the locking script (scriptPubKey) of the UTXO that we are trying to spend
- follow additional rules according to the SIGHASH flag

The possible SIGHASH flags, values and meaning are:

ALL (0x01) Signs all the inputs and outputs, protecting everything except the signature scripts against modification. This transaction is final and no one can modify it without invalidating this signature.

NONE (0x02) Signs all of the inputs but none of the outputs, allowing anyone to change where the satoshis are going. A miner will always be able to send the funds to his address. It can also be used to send funds somewhere as long as everybody else also sends funds; in this case it is assumed that one of the signers will use SIGHASH **ALL** (or **SINGLE**) to lock the outputs before broadcasting. It can be used as a blank check to a miner.

SINGLE (0x03) Sign 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. It can be used if someone creates a transaction with some inputs that they cannot spend and specific outputs and then send to other participants, one of which will to sign (with **ALL** or **SINGLE**), if they agree. A concrete example is provided later in this section.

ALL|ANYONECANPAY (0x81) Signs only this one input and all the outputs. It allows anyone to add or remove inputs, so anyone can contribute additional satoshis but they cannot change how many satoshis are sent not where they go. It can be used to implement kickstarter-style crowdfunding.

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 like. This input can be spend even in another transaction!

SINGLE|ANYONECANPAY (0x83) Signs this one input and its corresponding output. Allows anyone to add or remove other inputs. A potential use would be as a means to exchange colored coin[9] tokens with satoshis. For example, one can create a transaction that has an input which holds 10 of their tokens and an output of 10 million satoshis to an address that they own. This transaction would be invalid since the inputs do not provide the 10 million satoshis but it can be shared with others. If someone wants to claim the 10 tokens they can add an input with at least 10 million satoshis and an output that sends the 10 tokens to them. This would complete the transaction which can then be broadcasted.

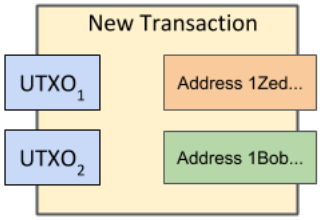


Figure 6.2: Example transaction with two inputs and two outputs.

For an example of **SINGLE**, consider creating the transaction shown above. Alice needs to pay 1.5 bitcoins to Zed and they agreed with Bob that he will contribute 0.5 of that amount. Then Alice creates a transaction with two inputs, UTXO1 that she owns (with 1 BTC) and UTXO2 that Bob owns (with 1 BTC) and a single output that sends 1.5 bitcoins to Zed. She signed UTXO1 with SIGHASH **SINGLE** and sends the incomplete transaction to Bob. Bob cannot choose a UTXO other than UTXO2 since it will invalidate Alice's signature of UTXO1. However, he is free to add other outputs so he creates an output that sends the remaining bitcoins (he agreed to pay only 0.5) to one of his addresses. He can then sign UTXO2 with SIGHASH **ALL** which will effectively finalize the transaction[10]. Note that:

- The sequence numbers of other inputs are not included in the signature, and can be updated.
- As already demonstrated above, 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.

[9] https://en.bitcoin.it/wiki/Colored_Coins
[10] No more input or output changes are allowed without invalidating this signature.

6.4 Pay to script hash (P2SH)

This section explains the rationale behind Pay to Script Hash (P2SH) type of addresses and demonstrates with code.

Multi-signature transaction output type

To demonstrate the advantages of P2SH we will first go through a simple use case using first P2MS, the legacy multisignature standard transaction output type, and then implement the same script with the P2SH standard transaction output type.

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
```

Indeed, this is the locking script of the multisignature standard transaction output type. If someone needs to send money (lock funds) to that output script then they need to know it! The company would need to send this script to all their customers that wish to pay them.

This is not practical since the whole script is recorded on the blockchain for every transaction and more importantly has privacy issues; the company is revealing the public keys that control the funds.

If you think about it the precise locking/unlocking script of the funds should not concern the customers at all. Only the company should know how to spend them.

Pay to script hash (P2SH)

P2SH is a type of transaction output (BIP-16[11]) that 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 value is the hash of the redeem script:

[11] https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki

```
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 as described in section 5.3 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 only need to disseminate this address to the company's customers. They can send funds oblivious to how these funds are locked. The company knows how they are unlocked so they can prepare the appropriate unlocking script.

As an example, to spend the funds, the company can create the following script:

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

It has two parts, the redeem script's unlocking conditions (which in this case are two of the signatures) plus the redeem script. Notice how the redeem script is revealed only when the company spends the funds; and it only reveals the two signatures, not all three.

Validation occurs in the following steps:

Initially, the scriptSig part is pushed into the stack and then the stack is copied for later use. The copy would look like:

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

Then the execution continues normally:

```
<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
```

If the hashed redeem script is equal to the 20-byte hash value the above script will result **OP_TRUE** in the top of the stack, which validates that the correct redeem script was passed. We can now proceed in validating the actual redeem script. We do this by using the copy of the stack, by first deserializing the top element (redeem script) and then executing normally:

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

If that also results to **OP_TRUE** then the funds can be spend.

Summary / Advantages

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

- P2SH allows us to create arbitrary redeem scripts; we can thus create quite complex scripts and not be limited to the few standard transaction output types.
- Reduces the size of the funding transactions typically resulting in saving blockchain sace.
- Increases privacy by hiding the locking conditions.

Example: create a P2SH address based on a P2PK script

As we have seen P2SH allows us to wrap arbitrary scripts hiding the script itself until it is spend. To demonstrate we will wrap a simple P2PK script and display the P2SH address that corresponds to that script.

```
from bitcoinutils.setup import setup
from bitcoinutils.keys import P2shAddress, PrivateKey
from bitcoinutils.script import Script


def main():
    # always remember to setup the network
    setup('testnet')

    #
    # This script creates a P2SH address containing a P2PK locking script
    #

    # secret key corresponding to the pubkey needed for the P2PK locking
    p2pk_sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')

    # get the public key
    p2pk_pk = p2pk_sk.get_public_key()

    # get the address (from the public key)
    p2pk_addr = p2pk_pk.get_address()

    # create the redeem script
    redeem_script = Script([p2pk_pk.to_hex(), 'OP_CHECKSIG'])

    # create a P2SH address from a redeem script
    addr = P2shAddress.from_script(redeem_script)
```

```
28      print(addr.to_string())
29
30  if __name__ == "__main__":
31      main()
```

The result is address **2MvzN3FntupGqY66FuGzoK9HFXqPFyMxfVU** which can be shared to anyone that wishes to send us some funds.

Example: spend funds from a P2SH address

Assuming that someone has sent funds to the P2SH address we just created let us spend it programmatically. Again, we will hardcode the UTXOs for brevity but we can always use the proxy object mentioned in section 6.2 to get the UTXOs programmatically.

```
1   from bitcoinutils.setup import setup
2   from bitcoinutils.utils import to_satoshis
3   from bitcoinutils.transactions import Transaction, TxInput, TxOutput
4   from bitcoinutils.keys import P2pkhAddress, P2shAddress, PrivateKey
5   from bitcoinutils.script import Script
6
7   def main():
8       # always remember to setup the network
9       setup('testnet')
10
11      #
12      # This script spends from a P2SH address containing a P2PK script
13      #
14
15      # create transaction input from tx id of UTXO (contained 0.1 tBTC)
16      txin = TxInput('7db363d5a7fabb64ccce154e906588f1936f34481223ea8c1f2c935b0a0c945b', 0)
17
18      # secret key needed to spend P2PK that is wrapped by P2SH
19      p2pk_sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')
20      p2pk_pk = p2pk_sk.get_public_key().to_hex()
21      # create the redeem script - needed to sign the transaction
22      redeem_script = Script([p2pk_pk, 'OP_CHECKSIG'])
23
24      to_addr = P2pkhAddress('n4bkvTyU1dVdzsrhWBqBw8fEMbHjJvtmJR')
25      txout = TxOutput(to_satoshis(0.09), to_addr.to_script_pub_key() )
26
27      # no change address - the remaining 0.01 tBTC will go to miners)
28
29      # create transaction from inputs/outputs -- default locktime is used
30      tx = Transaction([txin], [txout])
31
32      # print raw transaction
33      print("\nRaw unsigned transaction:\n" + tx.serialize())
34
35      # use the private key corresponding to the address that contains the
36      # UTXO we are trying to spend to create the signature for the txin -
37      # note that the redeem script is passed to replace the scriptSig
38      sig = p2pk_sk.sign_input(tx, 0, redeem_script )
39
40      # set the scriptSig (unlocking script)
41      txin.script_sig = Script([sig, redeem_script.to_hex()])
42      signed_tx = tx.serialize()
```

```
43
44      # print raw signed transaction ready to be broadcasted
45      print("\nRaw signed transaction:\n" + signed_tx)
46      print("\nTxId:", tx.get_txid())
47
48
49  if __name__ == "__main__":
50      main()
```

> As already mentioned, the first time we spend from the address, i.e. broadcasting this transaction, reveals the redeem script to everyone.

6.5 Segregated Witness (SegWit)

In this section we will briefly explain what Segregated Witness is, what it changes and how we can create segwit scripts programmatically.

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

To illustrate see figure 6.3 (i) for the original transaction that we used as an example in chapter 1. We can see that the unlocking script is part of the input that we are spending. In (ii) we can see the segwit equivalent transaction where the unlocking script (**scriptSig**) is empty and it is moved outside the transaction.

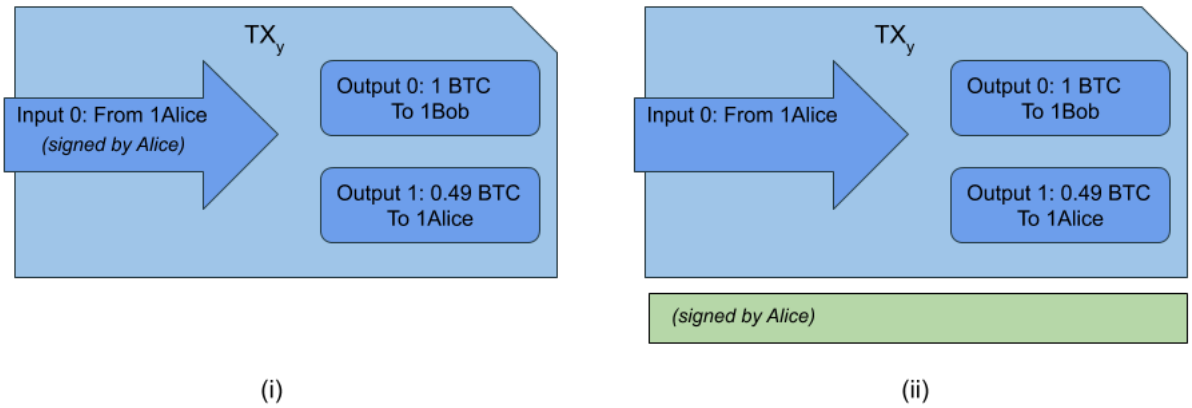


Figure 6.3: Example transaction with (i) a non-segwit input and (ii) a segwit input.

The segwit upgrade is described in detail in BIPs[12] 141, 143, 144 and 145 and it provides several benefits[13]. We will examine two of them here: *transaction malleability* and *effective block size increase*.

[12] Bitcoin Improvement Proposals
[13] https://bitcoincore.org/en/2016/01/26/segwit-benefits/

Transaction malleability

Each transaction is uniquely identified by its *transaction identifier* or **txid**. The **txid** is constructed by hashing[14] the serialized transaction (the blue part of figure 6.3).

It is possible to slightly change the unlocking script, e.g. by a miner, so as the resulting transaction is semantically identical to the original; thus still valid. This can be accomplished, for example, by slightly changing the structure of the signature[15].

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 of figure 6.3 (ii)) 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 counted as 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 block size increase of about 2.8x, if all transactions use segwit[16].

The *virtual* size, or **vsize** of a transaction is the size in bytes of a transaction including the segwit discount. For non-segwit transactions[17] **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:

[14] The SHA-256 hashing algorithm is applied twice.
[15] DER allows for some flexibility on how to encode the signature.
[16] E.g. block 748918 reached 2.77 MBs because it included a lot of P2WSH 2-of-3 multi-signature scripts.
[17] I.e. transaction with no segwit inputs.

```
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 section).

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>
```

The validation is executed as follows:

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**.

The **scriptCode** for P2WPKH is identical to P2PKH.

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 script*[18].

```
scriptPubKey: 0 3b892c61cc15f9a17<32 bytes>c1c722ff683b27f5a
scriptSig:
witness: 0 <signature1> <1 <pubkey1> <pubkey2> 2 CHECKMULTISIG>
```

The validation is executed as follows:

1. The '0' in scriptPubKey specifies that the following is a version 0 witness program.

[18] To clarify, witness script for a P2WSH is exactly what redeem script is for a P2SH.

2. The length of the witness program (32-bytes) indicates that it is a P2WSH type.
3. The witness must consist of the unlocking script followed by the 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*.

The **scriptCode** for P2WSH is the witness script.

Nested Segwit transaction output types

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)
```

And thus:

```
scriptPubKey: <HASH160 <20-byte-script-hash> EQUAL
scriptSig: <0 <20-byte-key-hash>>
witness: <signature> <pubkey>
```

Note that for spending[19] the **scriptSig** contains the redeem script as a single element and without any extra unlocking data before it since those are in witness (check section 6.4 for how P2SH works).

For validation the **scriptSig** is hashed with **HASH160** and compared to the **<20-byte-script-hash>** in **scriptPubKey**. If they are equal we can then verify the public key and signature as a native P2WPKH.

P2SH(P2WSH)

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

```
HASH160 (0 3b892c61cc15f9a17<32 bytes>c1c722ff683b27f5a)
```

[19] From a segwit-aware wallet.

And thus:

```
scriptPubKey: <HASH160 <20-byte-script-hash> EQUAL
scriptSig: <0 <32-byte-witness-script-hash>>
witness: 0 <signature1> <1 <pubkey1> <pubkey2> 2 CHECKMULTISIG>
```

Note that for spending the **scriptSig** contains the witness script as a single element and without any extra unlocking data before it since those are in witness (check section 6.4 for how P2SH works).

For validation the **scriptSig** is hashed with **HASH160** and compared to the **<20-byte-script-hash>** in **scriptPubKey**. If they are equal we can then verify the witness script as a native P2WSH.

Implemented as a 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 (see section 6.7) of the coinbase transaction. Being in the coinbase means that the witnesses are represented in the header via the **hashMerkleRoot** (see section 1.4 for a reminder).
- 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[20]; OP_0 <bytes in hex> will remain in the stack.

Example: spend a native segwit output type

Assuming that a P2WPKH address have some funds (UTXOs) we can use the following code to send some funds to a legacy address. As usual, we hardcode the UTXOs for simplicity.

```
1 from bitcoinutils.setup import setup
2 from bitcoinutils.utils import to_satoshis
3 from bitcoinutils.transactions import Transaction, TxInput, TxOutput
```

[20] If there are more than one elements after execution the stack is not *clean* and is considered non standard for legacy transactions. For segwit v0 (P2WSH) and Tapscript they are considered invalid.

```
from bitcoinutils.keys import P2pkhAddress, PrivateKey
from bitcoinutils.script import Script

def main():
    # always remember to setup the network
    setup('testnet')

    # the key that corresponds to the P2WPKH address
    priv = PrivateKey("cVdte9ei2xsVjmZSPtyucG43YZgNkmKTqhwiUA8M4Fc3LdPJxPmZ")

    pub = priv.get_public_key()

    fromAddress = pub.get_segwit_address()
    print(fromAddress.to_string())

    # amount is needed to sign the segwit input
    fromAddressAmount = to_satoshis(0.01)

    # UTXO of fromAddress
    txid = '13d2d30eca974e8fa5da11b9608fa36905a22215e8df895e767fc903889367ff'
    vout = 0

    toAddress = P2pkhAddress('mrrKUpJnAjvQntPgz2Z4kkyr1gbtHmQv28')

    # create transaction input from tx id of UTXO
    txin = TxInput(txid, vout)

    # script code required for signing; for p2wpkh it is the same as p2pkh
    script_code = Script(['OP_DUP', 'OP_HASH160', pub.to_hash160(),
                          'OP_EQUALVERIFY', 'OP_CHECKSIG'])

    # create transaction output
    txOut = TxOutput(to_satoshis(0.009), toAddress.to_script_pub_key())

    # create transaction without change output - if at least a single input is
    # segwit we need to set has_segwit=True
    tx = Transaction([txin], [txOut], has_segwit=True)

    print("\nRaw transaction:\n" + tx.serialize())

    sig = priv.sign_segwit_input(tx, 0, script_code, fromAddressAmount)
    tx.witnesses.append( Script([sig, pub.to_hex()]) )

    # print raw signed transaction ready to be broadcasted
    print("\nRaw signed transaction:\n" + tx.serialize())
    print("\nTxId:", tx.get_txid())


if __name__ == "__main__":
    main()
```

Example: spend a nested segwit output type

As described above we can also nest the new segwit address output types in a P2SH so that old non-segwit clients can send funds to new segwit-supporting clients. This example spends such a UTXO demonstrating a couple of additional details that need to be taken into

consideration.

```
from bitcoinutils.setup import setup
from bitcoinutils.utils import to_satoshis
from bitcoinutils.keys import PrivateKey, P2pkhAddress, P2shAddress
from bitcoinutils.transactions import Transaction, TxInput, TxOutput
from bitcoinutils.script import Script

def main():
    # always remember to setup the network
    setup('testnet')

    # the key that corresponds to the P2WPKH address
    priv = PrivateKey('cNho8fw3bPfLKT4jPzpANTsxTsP8aTdVBD6cXksBEXt4KhBN7uVk')
    pub = priv.get_public_key()

    # the p2sh script and the corresponding address
    redeem_script = pub.get_segwit_address().to_script_pub_key()
    p2sh_addr = P2shAddress.from_script(redeem_script)

    # the UTXO of the P2SH-P2WPKH that we are trying to spend
    inp = TxInput('95c5cac558a8b47436a3306ba300c8d7af4cd1d1523d35da3874153c66d99b09', 0)

    # exact amount of UTXO we try to spent
    amount = 0.0014

    # the address to send funds to
    to_addr = P2pkhAddress('mvBGdiYC8jLumpJ142ghePYuY8kecQgeqS')

    # the output sending 0.001 -- 0.0004 goes to miners as fee -- no change
    out = TxOutput(to_satoshis(0.001), to_addr.to_script_pub_key())

    # create a tx with at least one segwit input
    tx = Transaction([inp], [out], has_segwit=True)

    # script code is the script that is evaluated for a witness program type; each
    # witness program type has a specific template for the script code;
    # the script code that corresponds to P2WPKH is the same as P2PKH
    script_code = pub.get_address().to_script_pub_key()

    # calculate signature using the appropriate script code
    # remember to include the original amount of the UTXO
    sig = priv.sign_segwit_input(tx, 0, script_code, to_satoshis(amount))

    # script_sig is the redeem script passed as a single element
    inp.script_sig = Script([redeem_script.to_hex()])

    # finally, the unlocking script is added as a witness
    tx.witnesses.append(Script([sig, pub.to_hex()]))

    # print raw signed transaction ready to be broadcasted
    print("\nRaw signed transaction:\n" + tx.serialize())

if __name__ == "__main__":
    main()
```

6.6 Pay To Multi-signature (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 section 6.4 we have seen an example of an 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 in section 6.4 and send the funds there directly.

> 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.

6.7 Storing Data (OP_RETURN)

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.

Indirectly

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.

Let's go through one of the above examples, and explain how it would be possible to store data in the outputs of fake addresses. Remember that the output address is represented as the public key hash or 20 bytes (40 hex characters). Those can be faked to represent the data that we need to store. For example[21], to store "**3Nelson-Mandela.jpg?**" (20 characters) we would need to convert it to hexadecimal which is "**334E656C736F6E2D4D616E64656C612E6A70673F**" and would correspond to address "**15gHNr4TCKmhHDEG31L2XFNvpnEcnPSQvd**". Amazingly, the following outputs, if added together, contained an actual image of Nelson Mandela!

[21] Borrowed from Ken Shirriff's excellent blog post: https://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.html

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, but even if it wasn't such methods are not encouraged to say the least.

> 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!

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, called **OP_RETURN** specifically dealing with storing small amount of data on the blockchain.

The **OP_RETURN** opcode 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[22], the OMNI protocol[23] and verifiable PDFs[24].

An example script would be:

```
OP_RETURN 4f1edef24e9e2a169f56e1b0ae936d32232652dc51be1860ecd714
```

Example: Create a transaction with an OP_RETURN output

Creating an **OP_RETURN** output is quite straightforward. Remember that you will be spending bitcoins for the transaction fees and that there would most probably be a change output.

```
1 from bitcoinutils.setup import setup
2 from bitcoinutils.utils import to_satoshis
3 from bitcoinutils.transactions import Transaction, TxInput, TxOutput
4 from bitcoinutils.keys import P2pkhAddress, PrivateKey
5 from bitcoinutils.script import Script
6
7 def main():
8     # always remember to setup the network
9     setup('testnet')
```

[22] https://counterparty.io/
[23] https://www.omnilayer.org/
[24] https://verifiable-pdfs.org/

```
10
11      # create transaction input from tx id of UTXO (contained 0.01 tBTC)
12      txin = TxInput('ab48f4e23bf6ddf606714141ac87c3e921c8c0bebeb7c8abb2c799e9ff96ce6f', 0)
13
14      # create the OP_RETURN transaction output that contains "Hello!"
15      txout = TxOutput(to_satoshis(0), Script(['OP_RETURN', '48656c6f21']))
16
17      # create another output to get the change - note that we have hardcoded
18      # the fee; you should not :)
19      change_addr = P2pkhAddress('mmYNBho9BWQB2dSniP1NJvnPoj5EVWw89w')
20      change_txout = TxOutput(to_satoshis(0.008), change_addr.to_script_pub_key())
21
22      # create transaction from inputs/outputs -- default locktime is used
23      tx = Transaction([txin], [txout, change_txout])
24
25      # print raw transaction
26      print("\nRaw unsigned transaction:\n" + tx.serialize())
27
28      # use the private key corresponding to the address that contains the
29      # UTXO we are trying to spend to sign the input
30      sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')
31
32      # note that we pass the scriptPubkey as one of the inputs of sign_input
33      # because it is used to replace the scriptSig of the UTXO we are trying to
34      # spend when creating the transaction digest
35      from_addr = P2pkhAddress('myPAE9HwPeKHh8FjKwBNBaHnemApo3dw6e')
36      sig = sk.sign_input(tx, 0, from_addr.to_script_pub_key())
37
38      # get public key as hex
39      pk = sk.get_public_key().to_hex()
40
41      # set the scriptSig (unlocking script)
42      txin.script_sig = Script([sig, pk])
43      signed_tx = tx.serialize()
44
45      # print raw signed transaction ready to be broadcasted
46      print("\nRaw signed transaction:\n" + signed_tx)
47
48
49  if __name__ == "__main__":
50      main()
```

6.8 Exercises

Exercise 6.1 Write a program that spends a UTXO. The user will provide a P2PKH address as a parameter and he will then be prompted to choose between the available UTXOs of that address.

Exercise 6.2 In mainnet, how can we estimate what is an appropriate fee to include to a transaction?

Exercise 6.3 Write a scriptPubKey script that requires both a key and password to unlock.

Exercise 6.4 Create a P2SH address that corresponds to a 2-of-2 multisignature scheme. Display the address.

Exercise 6.5 Create a script that spends funds from the P2SH address created in the previous exercise.

Exercise 6.6 Write a Python function that goes through a serialized transaction and calculates what percentage of its size are the **scriptSig**s.

Exercise 6.7 Write a script that goes through a block and prints the percentage of all the **scriptSig**s relative to the size of the block.

Exercise 6.8 How can we calculate what the maximum *effective* block size limit is with segwit?

Exercise 6.9 Create a P2WSH address that contains a P2PK locking script. Then display the address.

Exercise 6.10 Create a transaction that spends UTXOs from a P2WSH address that contains a P2PK locking script.

Exercise 6.11 Create a P2SH(P2WSH) address that contains a P2PK locking script. Then display the address.

Exercise 6.12 Create a transaction that spends UTXOs from a P2SH(P2WSH) address that contains a P2PK locking script.

Exercise 6.13 Create a transaction that sends some bitcoins to a 2-of-2 P2MS standard output type.

Exercise 6.14 Create a transaction that spends some bitcoins from the 2-of-2 P2MS created above. Remember the **CHECKMULTISIG** bug.

Exercise 6.15 Create a 2-of-3 multisig (wrapped in P2SH) and display the address.

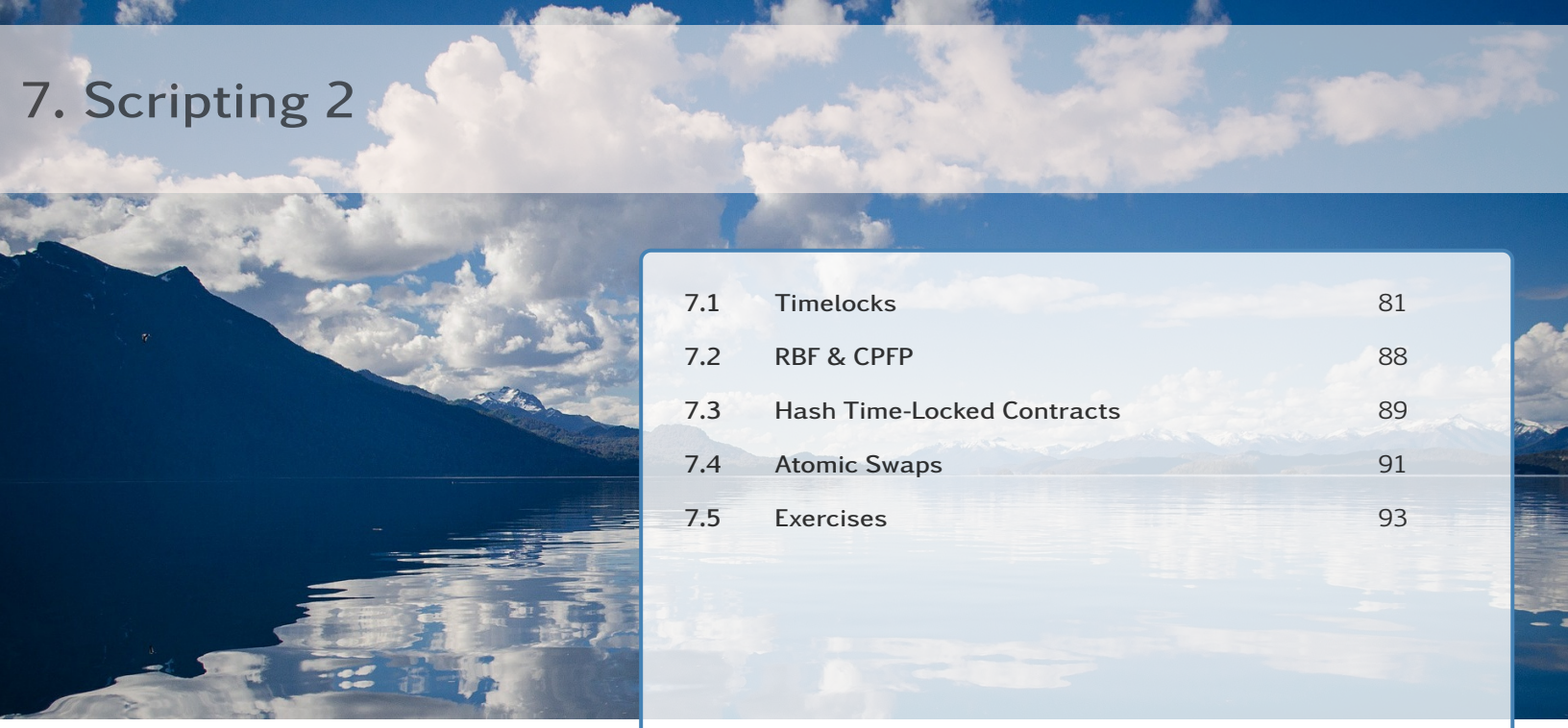
Exercise 6.16 Create a transaction that spends funds from the 2-of-3 multisig created above. Remember the **CHECKMULTISIG** bug.

Exercise 6.17 The Bitcoin white paper (PDF) is stored on the blockchain. The transaction id is:

54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713

Can you extract the data from this transaction and reconstruct the PDF?

7. Scripting 2



This chapter is build upon the previous one and continues to explore more advanced scripts and techniques for locking and unlocking funds in the Bitcoin network. Several examples are provided.

7.1 Timelocks

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.

Absolute at transaction 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:

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.

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

This can be achieved by setting **nLocktime** and **nSequence** appropriately.

```
nLocktime = current_best_height + 1
nSequence = 0xFFFFFFFE
```

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 higher sequence number transactions instead of more profitable ones (i.e. higher fee) which would never work.

For this reason the **nSequence** input field was repurposed to specify additional transaction semantics like timelocks. Typical transactions have an **nSequence** of **0xFFFFFFFF**.

For example, as we can see in figure 7.1, if we want to spent a UTXO of transaction TX_x in block Y (a block in the future, say 700000) we need to create a transaction that spends it but also set **nLocktime** to Y. Then this new transaction TX_{x+1} will be invalid until that block height.

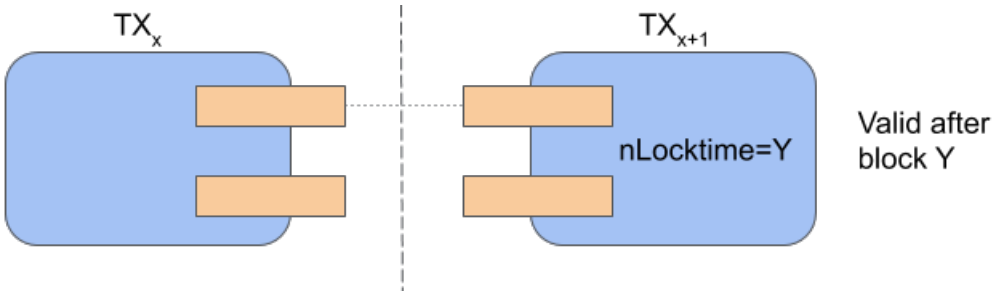


Figure 7.1: Example: absolute timelock.

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. This is problematic in several use cases and thus absolute timelocks at script level were created.

Absolute at script level

Absolute locktime is achieved at the script level using the **CHECKLOCKTIMEVERIFY** or *CLTV* opcode. In late 2015 the BIP-65[1] soft-fork redefined **OP_NOP2**[2] as **OP_CHECKLOCKTIMEVERIFY** allowing timelocks to be specified per transaction output. To spent the output, the signature and public key are required as usual but the nLocktime field of the spending transaction

[1] https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki

[2] A *no-operator* operator does nothing and is reserved to add future functionality.

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 an **expiry_time** *and* a P2PKH equivalent would look like:

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

To spend a transaction output with a timelock we need to specify the future block in **nLocktime** and activate the **nSequence** of the particular input that we want to spend to **0xFFFFFFFE**.

> Since a script with **CHECKLOCKTIMEVERIFY** becomes part of the blockchain it cannot be invalidated as described above with the transaction-level equivalent.

For example, as we can see in figure 7.2 we can create a locking script with CLTV on block Y (a block in the future, say 700000) 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 sets **nLocktime** to at least 700000. Then this new transaction TX_{x+1} will be invalid until that block height.

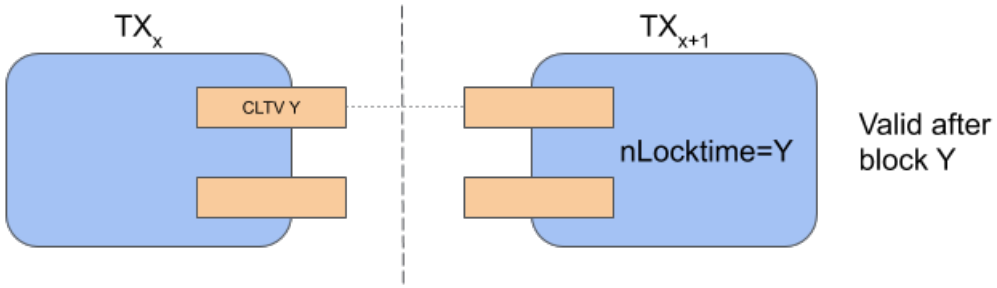


Figure 7.2: Example: script-level absolute timelock.

Relative at transaction input level

Relative timelocks where introduced in mid-2016 with BIPs 68[3] and 113[4] as a soft-fork that made use of the **nSequence** field of an input.

nSequence was repurposed 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 time) and the last 16 bits the value.

For example, as shown in figure 7.3, if we want to spent 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.

[3]https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki
[4]https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki

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.

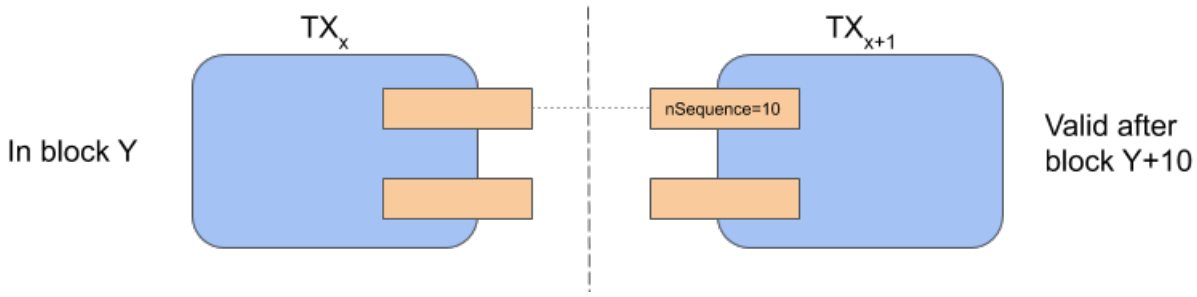


Figure 7.3: Example: relative timelock.

Relative at script level

The script-level equivalent of relative timelocks is using **CHECKSEQUENCEVERIFY** or *CSV* defined in BIP-112[5]. 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[6] and not the whole **nSequence** field.

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.

[5]https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki

[6]If timestamp the 23rd bit has to exist and be set. Also note that integers in the script should be serialized as signed integers in little-endian. Fortunately, programming libraries hide these details from the developers.

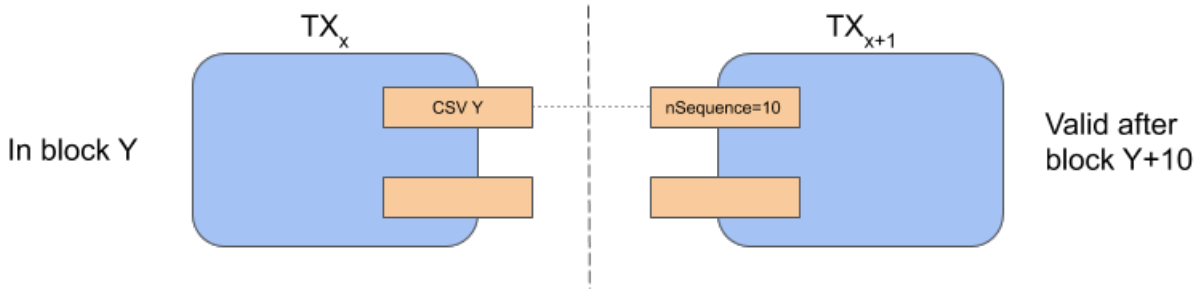


Figure 7.4: Example: script-level relative timelock.

Timelock types 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 (changle will) in between.
nLocktime + CLTV	Script	Absolute	Yes	Lock funds as part of a deal that allows no one access until 1-Jan-2020. Used in 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. Used in payment channels and Lightning network

Example: create a P2SH address with a relative timelock

Since the script we are going to create is not standard we need to wrap it using a P2SH output. Any funds send to that address will be locked for 20 blocks as well as with an P2PKH-equivalent script. The example is borrowed from **python-bitcoin-utils**[7].

```
1 from bitcoinutils.setup import setup
2 from bitcoinutils.transactions import Transaction, TxInput, TxOutput, Sequence
3 from bitcoinutils.keys import P2pkhAddress, P2shAddress, PrivateKey
4 from bitcoinutils.script import Script
5 from bitcoinutils.constants import TYPE_RELATIVE_TIMELOCK
```

[7] https://github.com/karask/python-bitcoin-utils/blob/master/examples/create_p2sh_csv_p2pkh_address.py

```
6
7  def main():
8      # always remember to setup the network
9      setup('testnet')
10
11     # This script creates a P2SH address containing a CHECKSEQUENCEVERIFY plus
12     # a P2PKH; locking funds with a key as well as for 20 blocks
13
14     # set values
15     relative_blocks = 20
16
17     seq = Sequence(TYPE_RELATIVE_TIMELOCK, relative_blocks)
18
19     # secret key corresponding to the pubkey needed for the P2SH (P2PKH) transaction
20     p2pkh_sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')
21
22     # get the address (from the public key)
23     p2pkh_addr = p2pkh_sk.get_public_key().get_address()
24
25     # create the redeem script
26     redeem_script = Script([seq.for_script(), 'OP_CHECKSEQUENCEVERIFY', 'OP_DROP',
27                             'OP_DUP', 'OP_HASH160', p2pkh_addr.to_hash160(),
28                             'OP_EQUALVERIFY', 'OP_CHECKSIG'])
29
30     # create a P2SH address from a redeem script
31     addr = P2shAddress.from_script(redeem_script)
32     print(addr.to_string())
33
34 if __name__ == "__main__":
35     main()
```

Example: spend funds from a P2SH timelocked address

Assuming that someone has sent funds to the P2SH address that we created above we can use this program to spend it. As usual and for simplicity, values are hardcoded and no change outputs are specified. The example is borrowed from **python-bitcoin-utils**[8].

```
1  from bitcoinutils.setup import setup
2  from bitcoinutils.utils import to_satoshis
3  from bitcoinutils.transactions import Transaction, TxInput, TxOutput, Sequence
4  from bitcoinutils.keys import P2pkhAddress, P2shAddress, PrivateKey
5  from bitcoinutils.script import Script
6  from bitcoinutils.constants import TYPE_RELATIVE_TIMELOCK
7
8  def main():
9      # always remember to setup the network
10     setup('testnet')
11
12     # We assume that some 11.1 tBTC have been send to that address and that we know
13     # the txid and the specific UTXO index (or vout).
14
15     # set values
16     relative_blocks = 20
17     txid = '76c102821b916a625bd3f0c3c6e35d5c308b7c23e78b8866b06a3a466041db0a'
```

[8] https://github.com/karask/python-bitcoin-utils/blob/master/examples/spend_p2sh_csv_p2pkh.py

```
18      vout = 0
19
20      seq = Sequence(TYPE_RELATIVE_TIMELOCK, relative_blocks)
21
22      # create transaction input from tx id of UTXO (contained 11.1 tBTC)
23      txin = TxInput(txid, vout, sequence=seq.for_input_sequence())
24
25      # secret key needed to spend P2PKH that is wrapped by P2SH
26      p2pkh_sk = PrivateKey('cRvyLwCPLU88jsyj94L7iJjQX5C2f8koG4G2gevN4BeSGcEvfKe9')
27      p2pkh_pk = p2pkh_sk.get_public_key().to_hex()
28      p2pkh_addr = p2pkh_sk.get_public_key().get_address()
29
30      # create the redeem script - needed to sign the transaction
31      redeem_script = Script([seq.for_script(), 'OP_CHECKSEQUENCEVERIFY', 'OP_DROP',
32                              'OP_DUP', 'OP_HASH160', p2pkh_addr.to_hash160(),
33                              'OP_EQUALVERIFY', 'OP_CHECKSIG'])
34
35      # to confirm that address is the same as the one that the funds were sent
36      #addr = P2shAddress.from_script(redeem_script)
37      #print(addr.to_string())
38
39      # send/spend to any random address
40      to_addr = P2pkhAddress('n4bkvTyU1dVdzsrhWBqBw8fEMbHjJvtmJR')
41      txout = TxOutput(to_satoshis(11), to_addr.to_script_pub_key() )
42
43      # no change address - the remaining 0.1 tBTC will go to miners)
44
45      # create transaction from inputs/outputs
46      tx = Transaction([txin], [txout])
47
48      # print raw transaction
49      print("\nRaw unsigned transaction:\n" + tx.serialize())
50
51      # use the private key corresponding to the address that contains the
52      # UTXO we are trying to spend to create the signature for the txin -
53      # note that the redeem script is passed to replace the scriptSig
54      sig = p2pkh_sk.sign_input(tx, 0, redeem_script )
55
56      # set the scriptSig (unlocking script) -- unlock the P2PKH (sig, pk) plus
57      # the redeem script, since it is a P2SH
58      txin.script_sig = Script([sig, p2pkh_pk, redeem_script.to_hex()])
59      signed_tx = tx.serialize()
60
61      # print raw signed transaction ready to be broadcasted
62      print("\nRaw signed transaction:\n" + signed_tx)
63      print("\nTxId:", tx.get_txid())
64
65
66  if __name__ == "__main__":
67      main()
```

Timelocks important caveat

Remember that **nLockTime** is set for the whole transaction and that it is either specified as a block height or as block header time (Unix Epoch). Thus, if the transaction tries to spend two inputs one with block height CLTV and the other with time CLTV it would be impossible to

spend.

```
Input 0:
        1 OP_CLTV
Input 1:
        500000001 OP_CLTV
```

The same would apply even for a single locking script that had both types of absolute locking.

```
1 OP_CLTV OP_DROP 500000001 OP_CLTV
```

Relative timelocks use **nSequence** which is per input and thus for CSV the issue comes up only when a single script has both types of locking.

7.2 RBF & CPFP

Replace-by-fee and *child-pays-for-parent* are two mechanisms that can help when transactions are stuck in the mempool due to low fees. They probably don't belong to the scripting section but RBF uses **nSequence**, the values of which, could influence timelocks.

Replace-By-Fee

Replace-by-fee, specified in BIP-125[9] is a mechanism for replacing any 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.

Similar to timelocks it applies only to version 2 transactions and you need to set **nSequence** to a value of **0x01** to **0x7fffffff**. However, since that such a value also enables 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 **0xfffffffd** which disables relative timelocks.

When creating transactions with the Bitcoin core software RBF transactions are opt-in. A replaceable transaction has the **bip125-replaceable** flag set to **yes** in its JSON display. You can make all transactions replaceable by default by setting **walletrbf=1** as a node option.

To work, in addition to setting the **nSequence** the transaction needs to reuse one or more of the same UTXOs[10] and increase the fees.

Using the Bitcoin core wallet there is an easy way to RBF a transaction (that is, of course, replaceable) by using **bumpfee**:

[9] https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki

[10] Note that this allows the new transaction to have different output addresses.

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

Child-pays-for-parent

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 (i.e. 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. There is nothing special with CPFP and can apply to any transaction. It makes use of miners' incentives to prioritize a transaction.

> The *sender* of some funds can use RBF to increase the fee to unstuck a transaction from the mempool. The *recipient* of some funds can use CPFP to indirectly increase the fee to unstuck a transaction from the mempool.

7.3 Hash 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, e.g. a passphrase, is publicly revealed. The passphrase can be shared by any means. All hashlock outputs using the same passphrase can then be spent. Such a locking script would be:

```
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).

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

HTLC

A *Hashed Time-Locked Contract* (BIP-199[11]) 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. For 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.

Let's go through this scenario in more detail. Alice wants to learn some information, e.g. a passphrase, and is willing to pay to get it. Bob has the passphrase and is willing to *sell* it[12]. To setup an HTLC, see figure 7.5 (i), the following steps are required:

- 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 now both create the script and P2SH address Alice sends funds to the new P2SH address

According to the locking script there are only two possible scenarios as seen in figure 7.5 (ii) and (iii) respectively.

(ii) Bob claims the funds and in doing so reveals the passphrase.

(iii) Bob does not claim the funds until the agreed timeout. Alice can take the funds back.

[11] https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki

[12] This scenario might sound very abstract but it will make more sense in the Atomic Swaps section.

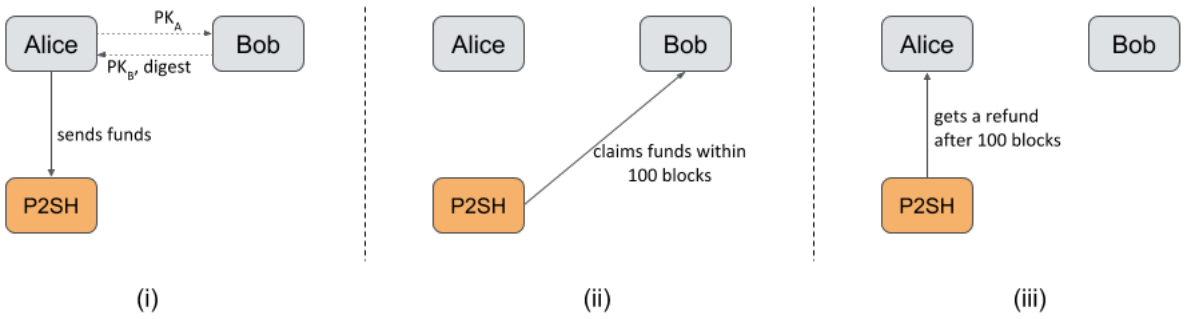


Figure 7.5: HTLC setup with potential scenarios.

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[13] and potentially several others.

7.4 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 *effectively* 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.

Let us describe a step-by-step example demonstrating the mechanism as seen in figure 7.6.

(i) Initial setup

- Alice and Bob exchange public keys on both Bitcoin and Litecoin.
- Alice and Bob agree upon the timeout thresholds, say 48 and 24 hours.
- Alice knows of a passphrase (pre-image) which is hashed to produce a digest; the latter is shared with Bob.

(ii) Both Alice and Bob create a HTLC for the Bitcoin and Litecoin blockchain respectively. The funds can be redeemed by:

- the passphrase and their counterpart's signature, or
- by both Alice's and Bob's signatures.
- Alice sends 1 BTC to the Bitcoin's HTLC and Bob sends 100 LTC to the Litecoin's HTLC. *No one* broadcasts the transaction!

[13] https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/

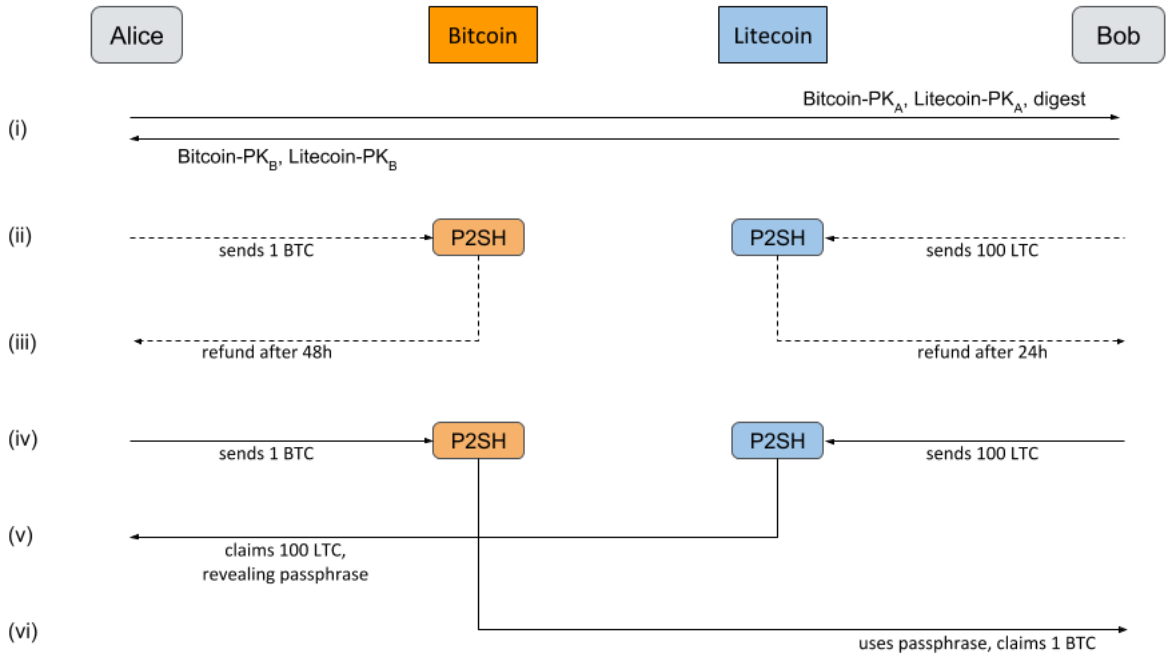


Figure 7.6: Atomic swap between Bitcoin and Litecoin blockchains.

(iii) Both Alice and Bob create a refund transaction for the funds they have just sent

- Alice creates a timelocked refund transaction that would be valid after 48 hours.
- Bob creates a timelocked refund transaction that would be valid after 24 hours.
- Both pass the refund transaction to their counterpart for their signature (satisfying the "both signatures" clause of the HTLCs).
- They now both have a final refund transaction that is ready to be broadcasted if something goes wrong.

(iv) Alice and Bob both broadcast the funding transaction from step (ii).

(v) Alice unlocks the Litecoin HTLC by the "passsphrase" clause of the HTLC revealing the passphrase in the process.

(vi) Bob uses the passphrase to unlock the Bitcoin HTLC as well and the atomic swap was successful.

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!

> Participants in an atomic swap need to inspect the blockchain for the relevant transactions!

Atomic swaps allow for trustless exchange between assets of different blockchains. One potential use case is trustless decentralized exchanges.

7.5 Exercises

Exercise 7.1 In section 7.1 we created an address for a relative timelock. Now create an address that locks the funds with an absolute timelock some time in the future (use block height or timestamp).

Exercise 7.2 In section 7.1 we spent from an address with a relative timelock. Now create a script that unlocks the funds from an address with an absolute timelock like the one created in the previous exercise.

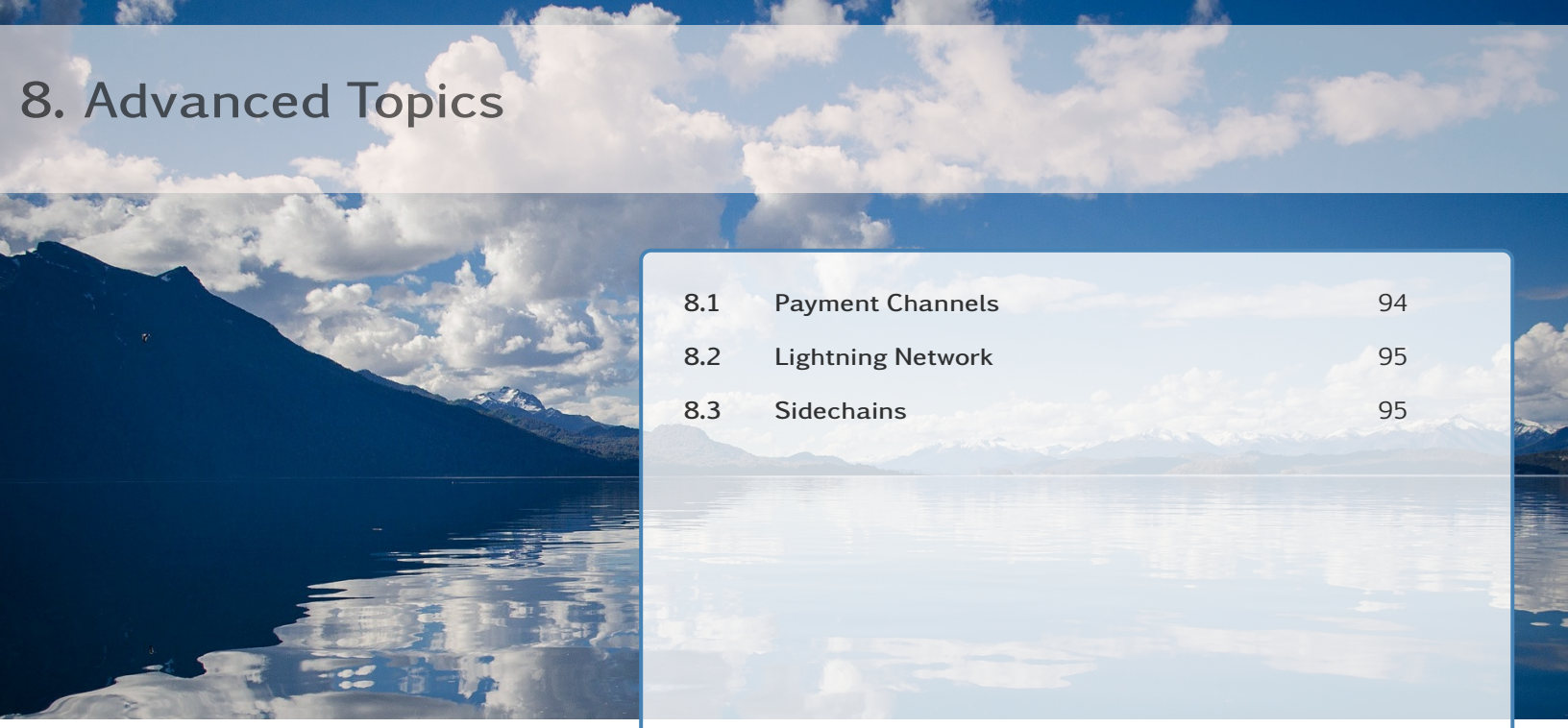
Exercise 7.3 Implement the example HTLC scenario of section 7.3. Create the appropriate scripts for both Alice and Bob.

Exercise 7.4 Write the HTLC locking script that Alice needs to create for the atomic swap in step (ii) as described in section 7.4.

Exercise 7.5 Write the unlocking script that Alice needs to use to claim the litecoin in step (v) as described in section 7.4.

Exercise 7.6 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.

8. Advanced Topics



8.1 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[1] 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_INPUT)

Spillman-style payment channels

TODO

CLTV-style payment channels

TODO

Poon-Dryja payment channels

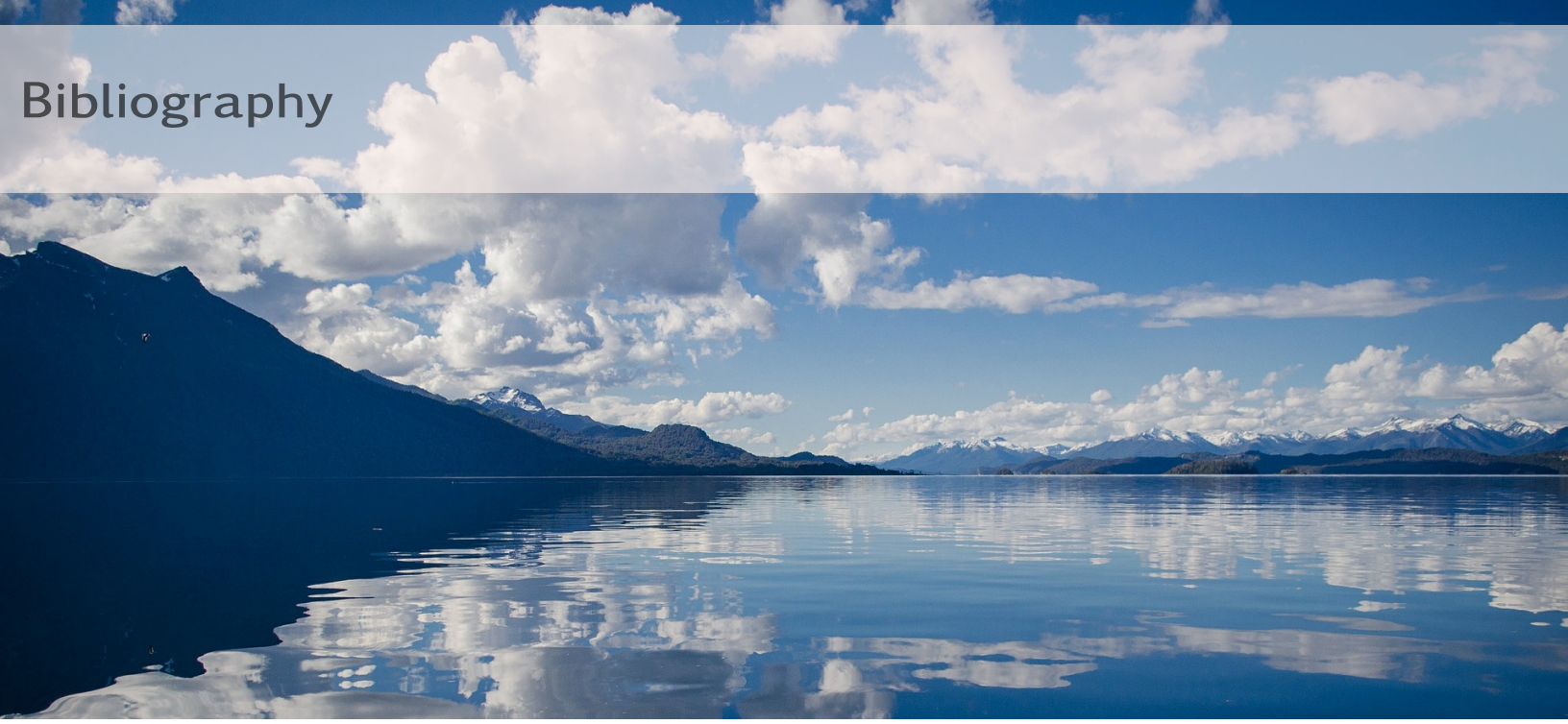
TODO

[1] https://en.bitcoin.it/wiki/Payment_channels

8.2 Lightning Network

8.3 Sidechains

Bibliography



[1] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. Tech. rep. 2002.