UMEÅ UNIVERSITY
Department of Computing Science
Master thesis report

Master degree project in computer science
30HP, 5DV143, VT19

Evaluating Cross-chain Settlement and Exchange in Cryptocurrency

Name	Carl-Johan Andersson
CAS	caan0156@umu.se
CS	c14can@cs.umu.se
Date	2019-04-16

Abstract
This is empty for now

University Supervisor
Jan-Erik Moström (jem@cs.umu.se)
Company Supervisor
Oskar Janssssson (oskar.janson@cinnober.com)

Glossary

Crypto currency - A currency backed not by centralized authorities but by mathematical evidence and clever mechanisms
Bitcoin - The very first and most mature cryptocurrency on earth
Litecoin - A alternative to Bitcoin, mostly a clone with a few changes
satoshi - The smallest fraction of a bitcoin. 100.000.000 satoshis = 1₿
Satoshi Nakamoto - The pseudonym used by the creator of bitcoin
Proof-of-work - A system where someone can prove mathematically that work was put into doing something.
Mining - In this context refers to looking for a valid hash in a proof-of-work system. Most often by checking random numbers in the nonce field until a valid hash is found
Onchain - Something that is onchain (for example a transaction) has been included in the blockchain and can be safely assumed to be immutable, in other words can't be changed.
Offchain - Something that is not on the blockchain.
Atomic (Adjective) - Something that only has two outcomes. Either completed fully or no changes to the state at all. An atomic task can not be half completed.
Alice and Bob - Alice and Bob are two fictional people used in examples.
Hash - A hash is a mathematical way of producing a one-way "fingerprint" of any data, easy to check but nearly impossible to reverse.
Pre-image - A pre-image is the name given to data as it was before being altered. For example Hash(D) = H, in this case D would be the pre-image of H

More to come...

Contents

1 Introduction **1**

1.1 The origins of Bitcoin . . . 1

1.1.1 Satoshi Nakamoto . . . 1

1.2 Basics on Bitcoin . . . 1

1.2.1 Digital signatures . . . 2

1.2.2 Blocks and the blockchain . . . 3

1.2.3 Proof-of-work . . . 3

1.2.4 Software . . . 5

1.2.5 Programmable transactions . . . 5

1.3 Payment channels . . . 5

1.3.1 Lightning network . . . 5

1.4 Atomic swaps . . . 6

1.5 The goal of the project . . . 6

2 Bitcoin and Smart contracts **7**

2.1 Bitcoin: a peer to peer electronic cash . . . 7

2.1.1 Network wide consensus . . . 7

2.2 *Side Bar* . . . 8

2.3 Elliptic-curve cryptography & ECDSA . . . 9

2.3.1 Secp256k1 . . . 9

2.3.2 Math on the elliptic curve . . . 9

2.3.3 Private and public key . . . 11

2.3.4 ECDSA . . . 11

2.3.5 Addresses . . . 12

2.3.6 Homomorphism . . . 12

2.4 Script 13
2.4.1 Complex operations 14
2.4.2 Valid and invalid scripts 14
2.4.3 Bug in Script 14
2.5 Transactions 15
2.5.1 Signatures and sighash 16
2.5.2 Pay to public key hash (P2PKH) 19
2.5.3 Pay to script hash (P2SH) 19
2.5.4 Timelock and sequence 20
2.5.5 Raw transaction 21
2.6 Segregated Witness 21
2.7 On-chain Atomic swaps 22
2.7.1 The contracts 22
2.7.2 The process 22
2.7.3 Atomicity 23
2.8 *Side Bar* 23
2.8.1 How the swap could fail 25
2.9 Payment channels 25
2.10 Off-chain atomic swaps 32

3 Implementation **33**

4 Comparison **34**

5 Conclusion & Discussion **35**

6 Future research **36**

Bibliography **37**

Chapter 1

Introduction

Although blockchain and other types of distributed ledgers are still in their infancy a growing number of people and companies are starting to see that they hold great promise, especially when it comes to financial technology, where things like trust and security is held to be very important.

1.1 The origins of Bitcoin

The most well known, developed and researched blockchain technology is Bitcoin. The mysterious nature of Bitcoins creator makes it hard to pinpoint how and when the idea was first thought up and when the development started, the most exact way would be to pinpoint at: `2009-01-03 18:15:05`[1], which is the timestamp in the very first block in the Bitcoin blockchain, however the white paper (**Bitcoin: A Peer-to-Peer Electronic Cash System**)[11] specifying the technical details circulated on cryptographic mailing lists as early as 31 October 2008, and the domain name `bitcoin.org` was registered 18 August 2008.[8]

1.1.1 Satoshi Nakamoto

The author name given in the white paper is `Satoshi Nakamoto`, this name is believed to be a pseudonym. Satoshi remained in the bitcoin community for a couple of years, regularly posting on the forum `bitcointalk.org` and keeping up with conversations in the mailing list. Those who have been interested in finding out Satoshi's real identity have analyzed his active time and language used. The findings were that Satoshi was most active during Western European day time, and he also used a lot of Anglo-colloquialisms such as "bloody hard" [4], and "flat" instead of apartment, so a popular theory is that Satoshi lived in Britain at least during this time.[8]

`April 23, 2011` was the last time anyone ever heard from Satoshi Nakamoto, in a mail to a fellow developer Mike Hearn he said "I've moved on to other things. It's in good hands with Gavin and everyone.".[6] Speculations on who Satoshi really is are still going strong even to this day, but Nakamoto's true identity is so far unknown.[8][9]

1.2 Basics on Bitcoin

Most people, even the layman with no blockchain experience, have at least heard of Bitcoin. But the exact details of how it works is not common knowledge. Described in a single sentence: Bitcoin is a currency where a communal ledger, that is shared between all participants in the whole world, dictates who owns what in terms of money or other assets.

The regular monetary system we are used to has a centralized authority, for example a central bank, who decides how much money is in circulation, who can transact with who etc. The monetary system presented by Bitcoin has no centralized authority, instead it relies on decentralized, trust-less verification.

These usually are the main concerns people have with distributed ledgers:

- How is spending someone else's money prevented?
- What prevents someone from "printing" more Bitcoins?
- How is spending the same money in different places in the world prevented?
- How is the consensus on the order of transactions reached?
- How do I interact with it?
- What type of transactions can you make?

These questions will be given ansers in the sections below.

1.2.1 Digital signatures

Bitcoin would not be possible at all without the underlying cryptographic mathematics. When it comes to proving ownership in Bitcoin ECDSA (Elliptic curve digital signature algorithm)[12] is used. Elliptic curve cryptography will not be covered in depth here but the basic idea is that you have a private and public key. The public key can be shared with anyone without danger, the public key can be derived from the private key, but not the other way around, this is something called trap-door mathematics.[12][7] This means that it is easy to go one way via equations. But going back is nearly impossible.

The reason for it not being completely impossible is because any potential attacker could always just keep guessing private keys until the right one is found. However with a sufficiently large private key, let's say 256-bits, and a computer that could check one billion billion (10^{18}) private keys every second (If such a machine could exist at all) it would still take $\frac{(2^{256}/10^{18})}{(60\cdot60\cdot24\cdot365)} \approx 3.67 \cdot 10^{51}$ years to check all possible private keys.

The most common transaction made on the blockchain is one where you "pay to" a public key. Who ever holds the private key paired with that public key can then via a mathematical equation prove that they hold the private key without actually revealing the private key.[5]

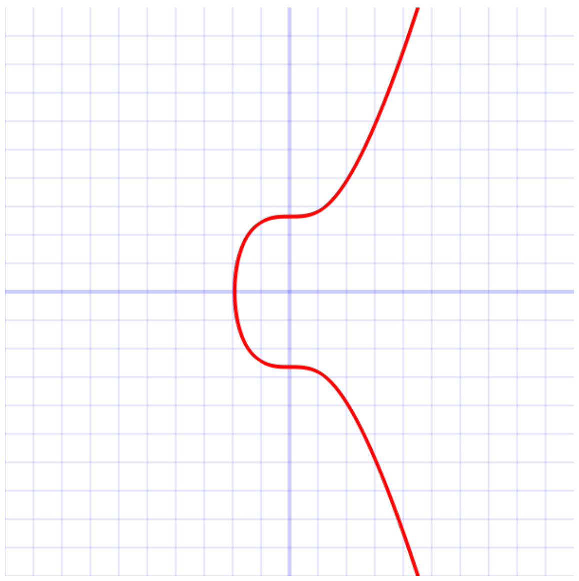


Figure 1.1: The `Secp256k1` plotted over real numbers. Note that the real curve is over a field, and thus looks more like a scattering of random points

The elliptic curve used can hold different parameters that defines it, certain elliptic curves are standardized and have their own names. The curve used in Bitcoin is named `Secp256k1`.[13][7]

The typical curve used in Elliptic curve cryptography is on the form $y^2 = x^3 + ax + b$. The `Secp256k1` is defined with $a = 0$ and $b = 7$, making the full `Secp256k1` equation: $y^2 = x^3+7$. Which is plotted in figure 1.1. For more in depth on elliptic curve cryptography see section 2.3

1.2.2 Blocks and the blockchain

A block is fairly simple to understand, it is simply a datatype or structure that holds information about itself, all the transactions that can fit and the previous block in the chain (more on that in a bit). Because every block holds information about the previous block you can follow all blocks backwards in time all the way back to the original, also called the genesis block.[1] This is what is referred to as the blockchain.

A block could be added to the chain by anyone in the world. However it will only be accepted if it has sufficient proof-of-work, and this is the key to how consensus is reached in the network.[7]

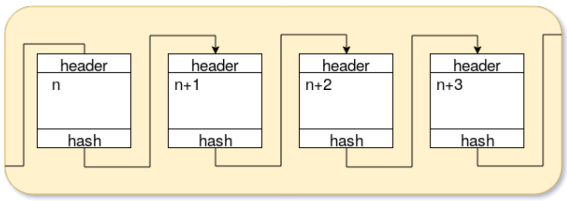


Figure 1.2: A basic overview of a blockchain

Block-header

Each block has a section of data called a header. The header contains meta-data about the block itself such as the version number, the id of the previous block in the chain, a timestamp of when the block was mined, the merkle root of all transactions (serves as proof of what transactions was included in the block) and a 32-bit field called nonce.

The size of the header is always 80-bytes, a blocks id[1] is equal to the hash of its header. For example the genesis block in the Bitcoin blockchain has the id:

`000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f`

Something that you will find with all block-ids is that they always have leading zeroes. This is a side-effect caused by mining and proof-of-work as explained in section 1.2.3.

1.2.3 Proof-of-work

Proof-of-work is, just like digital signatures, based in cryptographic mathematics. Before we go on, you first need to understand what a hashing algorithm does, the hashing algorithm used in Bitcoin is called `SHA256`. `SHA256` takes in data of any size and produces a sort of finger print of 256 bits.

For example the `SHA256` of the text "cool":

```
echo "cool" | openssl sha256
```

[1]Block id and block hash refers to the same thing. The terms might be mixed throughout the report.

Produces:

```
27c16ce7e3861da034af1bb356d6a4f38cb84fa65d51fa62f69727143b4c6b60
```

The text produced is actually bytes represented in a hexadecimal number system, in fact the entire string can be considered to be a very large number. Just like with the digital signature there is no known viable way that can take a hash and find what the original data that produced it was.

There is a term in Bitcoin called mining difficulty, or just difficulty. This is a large number, 256 bits to be exact. When you want to add a new block to the chain you have to do something called mining, this is a process where you change the nonce-bits in the block-header until the id of the block (Think of this as a number) is less than the target difficulty. The term hashpower refers to how many times the machine you are using to mine blocks can test a certain combination of variables per second, or H/s (Hashes per second).

The difficulty of mining a block is adjusted about every 2 weeks (every 2016th block to be exact). The difficulty is calculated by each individual node whenever the block count is a multiple of 2016, the nodes go by the timestamps in the blocks, and should thus all reach the same conclusion. The new calculated difficulty is the one all the next 2016 blocks have to reach to.[7]

The accepted order of transactions is the order going backwards from the latest block on the longest chain. The longest chain is always the one that the majority is mining towards. What it basically means is that as long as you trust 51% of the participants in the network you can also trust that the order of transactions is correct.[2] This mechanism prevents things like double spending the same money and holds a property called emergent consensus, which means that eventually the entire network will agree on the order of transactions.

In figure 1.3 is a diagram showing the longest chain, meaning the chain with most proof of work. The block in green is the latest block on that chain. The yellow block is what is known as a stale block, a block that is part of the chain but not part of the longest chain of blocks. Transactions in a stale block are not considered valid, and eventually they can be pruned (deleted) because they do not effect the future in any way. A stale block occurs if a new block is found in different parts of the network at almost the same time. Then the network will be split, each node tries to find the next block on the chain of whatever block they received first. In the case shown below the chain on the left "won" and the green block is now the longest chain.

The red block in figure 1.3 is what is known as an orphan block, that is, it has no known parent in the chain. This can happen if someone mines a block that is malformed or when building the chain for a new node and the blocks are received out of order.

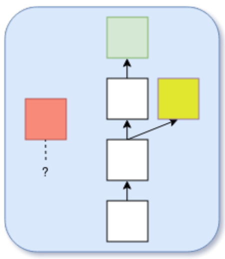


Figure 1.3: A diagram showing the longest chain, The green block is the latest block on the longest chain. The red block is a an orphan block, the yellow block is a stale block

[2]This is a simplification, but still holds true

1.2.4 Software

Just like how you can use regular money without knowing the underlying process and technology of banking systems for example, you can use Bitcoin without knowing the details of how it works. There is plenty of software that handles wallets, transactions etc for you.

1.2.5 Programmable transactions

Another feature that Bitcoin has, which makes it very versatile, is that transactions are programmable. As mentioned earlier the most common transaction is sending the money to someones public key. What really goes on here is that the person who wants to spend whatever money was sent to them has to prove programmaticlly that they own the private key related to that public key.

This is not the only type of transaction possible, Bitcoin has its own little programming language called `Script`. Any type of transaction that can be described in script is possible as a transaction. This is the basis for atomic swaps, payment channels and lightning network, which will be discussed in the following sections

1.3 Payment channels

One of the biggest problems facing Bitcoin is scalability. At the time of writing onchain[3] transactions is capped at about ≈ 7 T/s (transactions per second).[2] The throughput is limited by network propagation and a hard cap on block sizes, a block can only contain so many transactions before it is full. There are a couple of proposed solutions to this however, and one of them is connected payment channels

First off, a payment channel in Bitcoin is a type off trick using programmable transactions, were two users can open a bidirectional payment channel where an infinite number of transactions could be trustlessly exchanged without using the blockchain.

A channel can be opend by one or both participants by using a funding transaction. The funding transaction requires the signature of both participants to spend and is transmitted to the blockchain. After that the participants in the channel exchange commitment transactions that represent exchange in money. Any of the commitment transactions could be broadcast to the blockchain whenever a participant want to close the channel. Once a channel has been closed it cannot be used for further transactions. Mechanisms exists in the construction of these channels that makes it so one participant can't cheat the system and spend money that does not belong to them.

Payment channels alone were not enough to fix the scaling problem however, as a payment channel only allows two parties to exchange unlimited transactions. A proposed extension to the payment channels is the lightning network.

1.3.1 Lightning network

Lightning network is a relatively recent development in the Bitcoin community. How to construct payment channels has been known about for a while. But in January of 2016 a white paper was released detailing a promising new extension.[10] It showed that with a few changes to the Bitcoin protocol a new type of payment channel could be opened that allows transactions to propagate through multiple channels.[10]

[3]See glossary

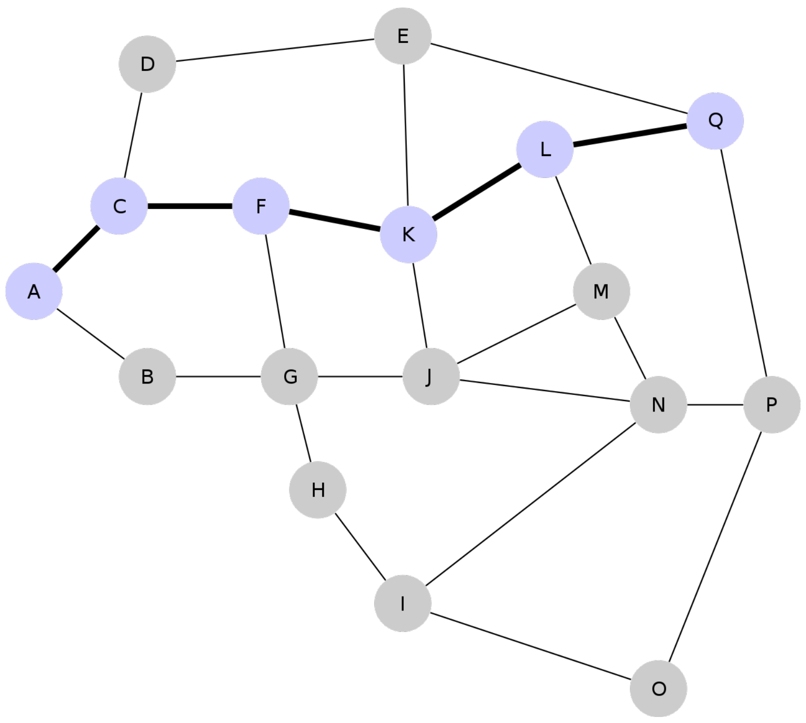


Figure 1.4: A basic overview of a Lightning network, each node represent someone and each edge represents a channel between two people.

Lightning network is really just a network of peers connected via payment channels. In figure 1.4 is an example. Let's say that Alice (node A) want to send a transaction to Quentin (node Q) but they have no direct payment channel between them. With lightning network they can send the transaction via the peers that are between them.

1.4 Atomic swaps

Another recent development in bitcoin is atomic swaps. Atomic swaps was first described by Tier Nolan in a post on `bitcointalk.org` on May 21, 2013.[3] A regular swap can for example be two parties exchanging currencies. Before, this could be done in person, or via a trusted third-party handling all the transactions. Atomic swaps however is as the name implies atomic. Meaning that they either go through completely or no assets change hands at all, they are also completely trustless meaning that you don't have to trust the other party or rely on any third-party

Just like lightning network atomic swaps use clever transaction scripts to achieve new functionality. Atomic swaps have been shown to be possible in over lightning network in theory. Yet today no standardized protocol exists, neither is it a part of the lightning network as it exists today.

1.5 The goal of the project

The main goal of the project is to experiment and perform as many types of atomic swaps as time allows with the purpose of developing a proposal on how future implementations of atomic swaps could be standardized, in regards of protocol and software.

A secondary goal of the project is to evaluate how the atomic swaps might fit into the modern financial system, which type is best suited for what purpose etc...

Chapter 2

Bitcoin and Smart contracts

This section is far from finished.

2.1 Bitcoin: a peer to peer electronic cash

As described in the title of the original white paper[11], Bitcoin is a peer to peer electronic cash system, which does not rely on any centralized third-party to verify the validity of any transactions or handling of transaction completion. Instead it is entirely decentralized and trustless. The mechanisms and mathematics that makes this possible are relatively recent discoveries in computer-science.

2.1.1 Network wide consensus

One of the main problems facing decentralized currencies before bitcoin was invented is the byzantine generals problem (Byzantine fault). This refers to independent agents in a system being unable to reach a consensus on what has transpired and what actions to take next. The problem can be imagined as the network being split, where one subsection of the network believes transaction order **A** is correct while another non-overlapping subsection thinks transaction order **B** is correct. How can this be resolved, and how will a new node joining the network know which order is the right one?

Bitcoin was the first cryptocurrency to properly solve this problem once and for all, with the help of something called proof-of-work. proof of work originates from the slightly older idea of hashcash.

Hashcash was/is a means to limit email spam and DDoS-attacks by a proof of work system. For example a sender could be required to produce a hash of message and nonce with a certain number of bits set to 0 at the start of the hash sequence. This (statistically) should take several attempts to produce. But correctness could be checked in a single step. Thus the hash sent together with the message could be considered proof of work. Because there is no known way to produce such a valid hash of a message without trying random combinations of bytes. Thus the only realistic way to have such a hash is if you worked for it. Looking for a valid hash in this kind of proof of work systems is often referred to as **mining**.

Bitcoin used this proof of work concept for another purpose however. Rather than combating spam the proof of work mechanism is used for reaching consensus. When ever a new block is added to the chain it needs to meet a certain proof of work requirement, called difficulty. This difficulty is set so that it should take the combined hashing power of all participants on average 10 minutes to find a valid hash of the next block in the chain.

There is a term called longest proof of work in bitcoin, this refers to the chain of blocks that has the most hash-power supporting it. The chain with the most work done is statistically the one with the most participants. As long as 51% or more of the participants in the network are honest the longest chain can be trusted. So any new members can accept the chain with most work as the truth.

Splits

Contention for the longest chain can arise if a new block is found in two different parts of the network at (almost) the same time. This is not a problem and will eventually be resolved. If you imagine two blocks (**A1** and **B1**) being mined in different parts of the network with the same parent block and half the network got **A1** first and the other half of the network got **B1** first. While the entire network will accept all valid blocks they will only mine towards continuing the chain on the block they received first. So if a new block **A2** with the parent block **A1** is found first the chain formed by block **B1** will be considered invalid and the network continues the chain on the **A** side.

Such contention is called a split, or a fork, and happens naturally once every week or so. As explained they will eventually resolve themselves. The split becomes increasingly more unlikely to survive the longer it goes on. To begin with it is unlikely that a new block will form a split in the first place, and for the split to survive another block both new chains will have to receive a new block at almost the same time.

In common bitcoin lingo there is no difference between a split and a fork. But in this report a **split will referred to as one occurring unintentionally** and **a fork being intentional** just so there is no confusion

How long was the longest split?

The longest splits that occurred by chance were 4 block long and has occurred at least at 3 different occasions.

The longest split ever was caused by an update to the bitcoin core reference implementation (**0.8.0**) that rejected a block that the other implementations did not reject, the nodes accepting the new block keept building on it while those who had updated built on a different chain. The split lasted for 52 blocks before it was resolved.

Forks

Forks happen when there is a disagreement in the bitcoin community when it comes to protocol and consensus. The most famous fork in all of cryptocurrency occurred on `1st August 2017` and was caused by a conflict regarding the size of blocks. How bitcoin will scale to world-wide use has always been a hot debate in the bitcoin community, the majority seeks to scale bitcoin via second layer solutions such as payment channels, lightning network and side-chains. However there were those who disagreed, and instead wanted the network to handle more transactions by making the blocks larger at cost of centralization.

A change in the blocksize requires the entire network to upgrade to the new protocol rules, but the block size increase was not accepted by enough nodes. So the group decided that a fork was the only way to resolve the issue. So as mentioned on `1st August 2017` the first block was mined on the new chain.

The new chain got the name **Bitcoin cash** (bcash) while the main chain still is called just **Bitcoin**. Most miners and node owners stayed with Bitcoin, but a fraction jumped ship and started working on bcash instead. Today both chains runs along side each other, co-existing.

Forks are considered a valid way to vote on what consensus rules and protocol changes should be made. When a change is planned to be made in the chain those who disagree can branch off and follow the old rules. Thus in a way everyone gets their way. You could even branch off on your own forked-chain alone. The main problem for those forking is that most vendors and users still consider the largest (most users, most developers, largest price, etc...) chain to be the valid one.

Soft and hard-forks
TODO

2.3 Elliptic-curve cryptography & ECDSA

As covered in the introduction, Elliptic-curve cryptography (**ECC**) and ECDSA is a fundamental building block of bitcoin. Elliptic curve cryptography relies on intractability of calculating the discrete logarithm of a elliptic curve element with respect to a publicly known base point. Or put another way: It is easy to calculate elliptic curve multiplication with multiplicand n. But calculating n from the resulting point is considered infeasible with sufficiently large curves and multiplicands.

An elliptic curve is defined by the equation $Y^2 = x^3 + ax + b$ and six domain parameters $E(p, a, b, G, n, h)$. **p** is the field that the curve is defined over, this is usually a very large prime number. The curve being defined over a field simply means that the points on the curve fall within $[0, p]$ rather than within the real numbers $\mathbb{R}$. In other words the curve is defined over the field $\mathbb{F}_p$. **a** and **b** are whatever number you put into the equation. **G** is the generator point, that is the point on the curve that will be used in point multiplication later. **n** is the order of G. What that means is that n is the largest number that G can be multiplied by before a point at infinity is produced. n pretty much tells you the limit on how points on the curve that can be generated from G. **h** is the co-factor of the curve. It can be calculated as follows: $h = \frac{1}{n}|(E(\mathbb{F}_p)|$, where $|(E(\mathbb{F}_p)|$ is the order/cardinality of the group of points possible on the curve over field $\mathbb{F}_p$. n is derived from G, G and p should be chosen in such a way that $h \leq 4$, preferably $h = 1$.

These domain parameters can be chosen manually or you can use predefined parameters. Elliptic curves that used predefined domain parameters are called named-curves. The named curve used by Bitcoin is called `Secp256k1`

2.3.1 Secp256k1

`Secp256k1` is defined with the following domain parameters (hexadecimal):

$p =$ `FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F`
or alternatively:
$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

$a = 0$
$b = 7$

$G =$ (`79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798`,
`483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8`)

$n =$ `FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141`
$h =$ `1`

2.3.2 Math on the elliptic curve

Two mathematical operations needs to be defined to operate on the elliptic curve: addition and multiplication

Point addition

Let's say you have to distinct points P and Q that both fall on curve $E(p, a, b, G, n, h)$ ($Y^2 = x^3 + ax + b$).

$$P + Q = R \Rightarrow (X_P, Y_P) + (X_Q, Y_Q) = (X_R, Y_R)$$

$$X_R = \lambda^2 - X_P - X_Q$$
$$Y_R = \lambda(x_P - X_R) - Y_P$$

where λ:

$$\lambda = \frac{Y_Q - Y_P}{X_Q - X_P} \mod p$$

Point multiplication

If P and Q are coincident, meaning that they have the same coordinates the equation is slightly different.

$$P + Q = R \Rightarrow P + P = R \Rightarrow 2P = R$$

This could be seen as P being multiplied with scalar 2. Most of the equation is the same as with addition, the difference is that:

$$\lambda = \frac{(3X_P^2 + a)}{(2Y_P)} \mod p$$

Faster multiplication with large scalars

Take $xP = R$ that could be calculated by summing P x times:

$$\sum_{n=1}^{x} P = R$$

This might work fine for smaller numbers but for a very large number, like $x = 2^{100}$ it will take infeasible amount of time to calculate. Luckily there is a convenient short cut that you can take called double and add.

First remember that: $P + P = 2P \Rightarrow 2P + P = 3P \Rightarrow 4P = 2(2P) \Rightarrow 8P = 2(2(2P))$

Lets say $x = 200$ in binary terms this could be written as $x = 128+64+8$ or $x = 2^7+2^6+2^3$ thus $200P = R$ could be written as

$$2^7P + 2^6P + 2^3P = R$$

which could be shorten to:

$$2(2(2(2(2(2(2P)))))) + 2(2(2(2(2(2P))))) + 2(2(2P))$$

which looks cumbersome but now instead of 200 calculations you only have to do 18.

2.3.3 Private and public key

Just as RSA cryptography, ECC relies on public-private key encryption and signatures. The public key can be shared freely to everyone, while the private key should, as the name implies, be kept private. Each unique private key has a corresponding public key, through mathematics it can be proven that someone holds the private key paired with a certain public key, without actually revealing the private key.

In ECC **a private key is a really large number**. Imagine you have curve $E(p, a, b, G, n, h)$ and you want to generate a brand new private key k. k could be any number between 0 and n. Any $k > n$ will produce the exact same public key so that will not work. **A public key in ECC is represented by a point in 2D space**, more specifically a point that falls on the curve. To generate a public key P from a private key k you perform $kG = P$ as described in the section above.

Compressed key

The public key is quite large, with two 256-bit numbers representing coordinates. But there is a clever trick we can use to compress the size of the key. Take the Secp256k1 curve for example ($Y^2 = x^3 + ax + b$). It is mirrored around the x-axis, meaning that for each x value there are two possible y values. Thus a public key can be represented by only it's x value plus a prefix telling you which resulting y-value to choose.

Note that because y and x is over $\mathbb{F}_p$ there is no negative value, instead the y value is referred to as even or odd.

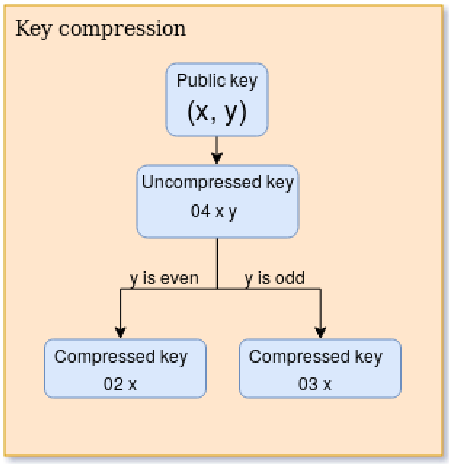


Figure 2.1: How to compress the public key in ecc

2.3.4 ECDSA

The main usage of ECC in cryptocurrency is for proving ownership of coins. The proof relies on elliptic curve mathematics like before. Lets say **Alice** has a message m and want to send it to **Bob** and also prove that the message came from her. First of let's establish some variables: k_A is the private key belonging to Alice, from that private key P_A was generated (The public key), that Bob knows about.

Signing

First calculate the hash of the message:

$$e = HASH(m)$$

If e has a bit-length (numbers in binary representation) that is longer than the bit-length of order n of the curve used. e has to be trimmed down so that the bit-lenghts match

Select a cryptographically-secure random number z that falls in the range $[1, n-1]$ and calculate a new curve point: $(x_1, y_1) = z \times G$

Calculate r and s such that: $r = (x_1 \mod n)$ and $s = (z^{-1}(e + r \times k_A) \mod n)$. if either r or s ends up being 0, generate a new z and try again.

The signature will be the point $(r, s) = S_A$.

Signature validation

If **Bob** wants to verify that it was actually **Alice** that signed the message m. He first has to do a sanity check on the signature S_A to make sure that it is a valid point on the curve and that s and r is within the range $[1, n-1]$ etc...

Calculate the hash e of m the same way as it was done during the signing process. Calculate

$$w = (s^{-1} \mod n)$$

and

$$u_1 = (ew \mod n)$$

$$u_2 = (rw \mod n)$$

From u_1 and u_2 calculate the point $(x_1, y_1) = u_1 \times G + u_2 \times S_A$

The signature is valid if and only if $r = x_1 \mod n$

2.3.5 Addresses

While in using Bitcoin normally you don't see the public keys directly, instead you mostly see something that is called addresses. The address is what you use when you want to send money to someone (see section 2.5.2). What the address really is, is just a representation of their public key.

To transform a public key to an address you first hash it using `SHA256` then that result with `RIPEMD160`, this entire process is called HASH160. After doing HASH160 you encode the resulting bytes in Base58, this is your address.

For example:

Compressed public key:
`03b2319bf63ca8959794f79056283150f1b49d577b2c48bd9e4a18d616e6f99bb4`

Hash160 of public key:
`76a9147ecfe7db089f521c3f49de0ee866f8e0fee97d1788ac`

Base58 of public key hash (Address):
`ms5UVmKaaz8kiL7DbMA1NG3t6T4C4GmGzG`

2.3.6 Homomorphism

Elliptic curves has a attribute called homomorphism. It will not be covered in depth, but what it can be used for is that two public keys (P_1, P_2) can be combined to form a third public key (P_R). If P_1 and P_2 was generated with corresponding private keys k_1 and k_2 then they can be combined to form the private key that generates P_R (k_R).

There are several ways of calculating a a new homomorphic key, this is the one used in later implementations:

Let's say you have private key k with corresponding public key P. And another temporary private key k_c with corresponding public key P_c. To generate the homomorphic public key (P_h) you can use the following equation (All the operations in equation 2.1 below are in elliptic-curve addition and multiplication, The second (2.2) uses regular operators):

$$Generate(P, P_c) = P * SHA_{256}(P||P_c) + P_c * SHA_{256}(P_c||P) = P_h \tag{2.1}$$

You now have Public key P_h. To figure out the Private key correlated with this public key you do the following equation:

$$Resolve(k, k_c) = k * SHA_{256}(P||P_c) + k_c * SHA_{256}(P_c||P) = k_h \tag{2.2}$$

Revocation keys

Imagine Alice and Bob wanting a public key that neither hold the private key to, that could at a later date be revealed. Alice holds the key set (k_A, P_A), and Bob has the set (k_B, P_B). Bob creates another key set (k_T, P_T) and sends P_T to Alice.

Alice generates a a new public key (P_R) from P_A and P_T with equation 2.1. Now the public key P_R could be used for whatever purpose needed, even though neither party knows the private key k_c at this point as calculating it requires both k_A and k_T and neither party holds both. At a later date Bob could reveal k_T to Alice, she could then use equation 2.2 to determine k_R.

This method is used later in a concept called revocable deliveries, see section 2.9. In that case (k_T, P_T) key set would be called commitment point keys, and (k_R, P_R) would be called revocation keys

2.4 Script

Script is the name of the programming language used in Bitcoin and its derivatives. It was not used to write the Bitcoin implementation but rather it is what makes transactions in Bitcoin so versatile. This section however will not focus on where or how Bitcoin uses Script, but rather on how Script itself functions.

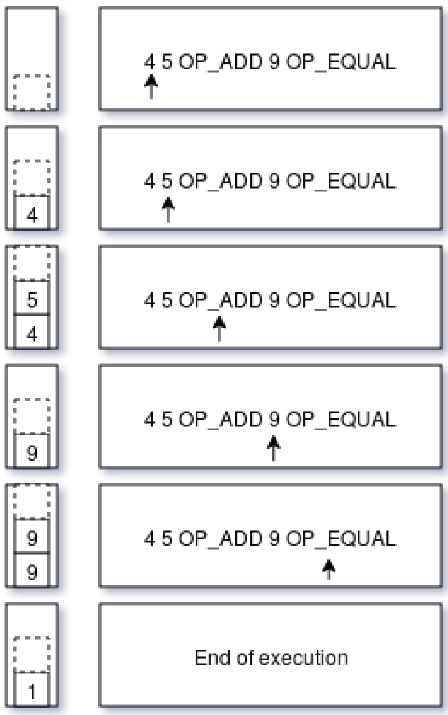


Figure 2.2: Execution of simple program, to the left is the stack, to the right is the program with execution pointer

Script is a forth-like stack based language that is not turing-complete. Turing-completeness means that a language can do anything that the imaginary turing-machine could do, in other words it basically means is that the language can do any mathematically sound operation. Script is **NOT** turing-complete on purpose, more on why can be found in the section about transactions (2.5). A good example is loops, in most languages there is some sort of structure that allows for a piece of code being executed repeatedly. In Scipt this is strictly disallowed, as it has neither for-loops or while-loops.

As mentioned earlier. Script is a stack-based language. That means that as the language executes it uses a stack to store data and variables. Do not confuse the term with the heap and stack from regular programming language discourse. In script there is no heap, instead the stack is the only form of memory, and it acts just as you would expect from a stack, to add a value you have to push it to the stack and to read a value you have to pop it from the stack.

The language it self is quite basic. It relies on operation codes (op codes) the size of a single byte. Most operations pop values from the stack, does something with the values, then pushes the result back on the stack. When Script is written out on paper many op codes and values are excluded, because they are implicit. For example:

```
OP_PUSHDATA1 4 FFFFFFFF
```

This operation pushes 4 bytes to the stack, the bytes all have the hexadecimal value `FFFFFFFF`. But usually when this operation is written out it is shorten to just:

```
FFFFFFFF
```

Whenever a hexadecimal value appears in the code it is implicit that that value is pushed to the stack in the form of bytes. Here is annother program:

```
4 5 OP_ADD 9 OP_EQUAL
```

This simple program will push 4 and 5 to the stack, `OP_ADD` pops two values from the stack (4 and 5) adds them together and pushes the result back on the stack. Then a 9 is pushed to the stack. `OP_EQUAL` pops two values form the stack and pushes a 1 if they were equal, 0 otherwise. In this case a 1 will be left on the stack as $4 + 5 = 9$. Figure 2.2 visualizes what happens at each step of execution.

2.4.1 Complex operations

There are some operations in Script with slightly higher complexity, which do not act like the others. One of them is `OP_VERIFY`, this will perform a verify check on the script. It will pop one value from the stack, and if that value equates to false execution will end immidietly and the entire script will be marked as invalid (see section 2.4.2). If the value equals true execution will continue where it left off. Some operations are combinations with the verify check, like for example `OP_EQUALVERIFY`. This is equal to writing `OP_EQUAL OP_VERIFY`, meaning that it first does an equal check then verifys the result.

There are several operations for checking signatures. These are not so complex in terms of what they do in the script. Their implementation is quite complex however. They reliy on outside information that is not present in the script to check the validity of a signature. The two most used are `OP_CHECKSIG` and `OP_CHECKMULTISIG`. These will not be covered in full in this section as they are more related to transactions so see section 2.5 for a full explanation.

2.4.2 Valid and invalid scripts

At the end of execution a script is marked as either valid or invalid. A script is invalid for the following reasons:

- The stack is empty
- There are more than one values on the stack
- The only value on the stack equates to false
- A VERIFY check fails sometime during execution.

A script is valid if and only if there is one value on the stack, and it is not equal to false.

2.4.3 Bug in Script

An interesting bit of trivia is that there is a bug in the language implementation. More specifically with the operation `OP_CHECKMULTISIG`. The bug makes it so this operation pops one more value from the stack than it is supposed to. This was not discovered until the network had been running for a while. It can't be easily fixed as it is now a part of the consensus rules. All implementations of Bitcoin node has to implement the bugged version of this op code otherwise consensus on the validity of transactions will break.

A fix to this bug would require a hardfok, see section 2.1.1, and has so far been considered not worth it.

2.5 Transactions

Transactions in Bitcoin are not as straight forward as you might expect a transaction to be. A transaction contains a list of inputs and a list of outputs as well as some metadata like version number and lock-time.

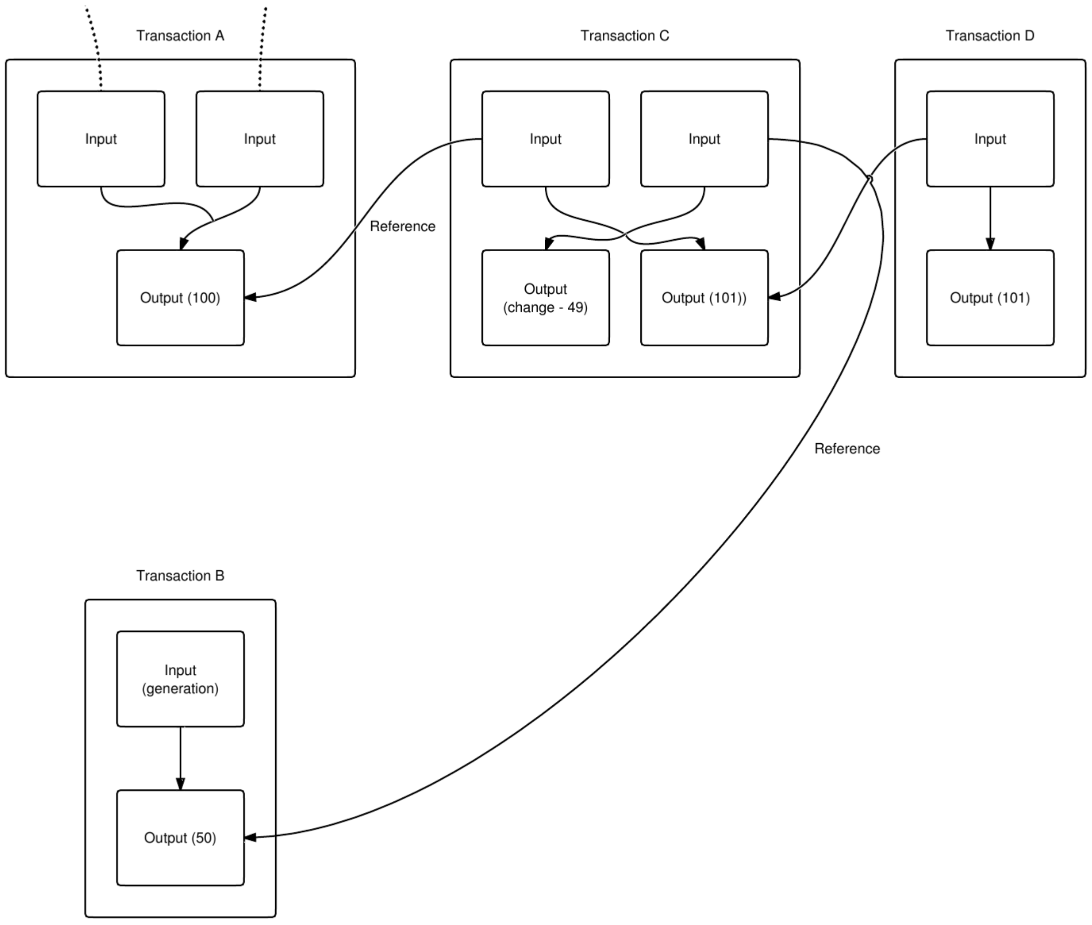


Figure 2.3: 4 example transactions and how inputs are connected to outputs

In simplified terms an output could be seen as the destination of a transaction, in other words it says how much and to whom the transaction is sent to. An input is a reference to a previous output. The inputs take the money from the outputs they reference and that money is used to fun the new outputs.

The inputs and outputs is where Script comes into the picture. Both outputs and inputs contains an incomplete script, together however they complete the script. The script in an output could be seen as a challenge, and the script in the input is the response. When a transaction is tested for validity the input script is appended to the script in the output and is executed. If the script comes out as valid the transaction is also valid. Here is a basic example: Let's say Alice wants to send a transaction to whoever can answer the very complex equation $4 + 3$. Her transaction output would contain the script:

```
4 3 OP_ADD OP_EQUALS
```

If this is executed as is it is invalid. But let's say Bob knows the answer to the equation he can then create a new transaction where the input contains the script:

```
7
```

Just as before this script is not valid by itself. But then the transaction is checked for validity the input will be appended to the start of the output script forming the following:

```
7 4 3 OP_ADD OP_EQUALS
```

Which is a valid script, thus bobs new transaction is also valid and he may spend the money as he see fits.

Obviously most transactions on the Bitcoin blockchain are not this simple. The most common form of transaction contians a script called P2PKH which stands for Pay to public key hash. Before we can go into details on this one however we first need to know about how signatures and sighash work in script and transactions.

2.5.1 Signatures and sighash

Section 2.3 covers public keys and signatures in depth.

Perhaps the most important operation in script is the `OP_CHECKSIG` operation and its cousins. `OP_CHECKSIG` pops two values from the stack, if the script is correctly implemented these two values should be the public key and a signature that was signed with the private key that was used to create the public key.

The question is: what is signed when the signature is created? Broadly speaking it is the hash of the transaction that is trying to spend the output, this is not entirely correct however. Appended to the signature that is a flag called **sighash**. The value of sighash tells the script interpreter what hash was signed during the creation of the signature. There are 4 types of sighash implemented:

SIGHASH_ALL
This can be considered the default sighash, if it is not specified it can safely be assumed that this type was used. This simply means that the entire transaction is signed with all outputs and all other inputs.

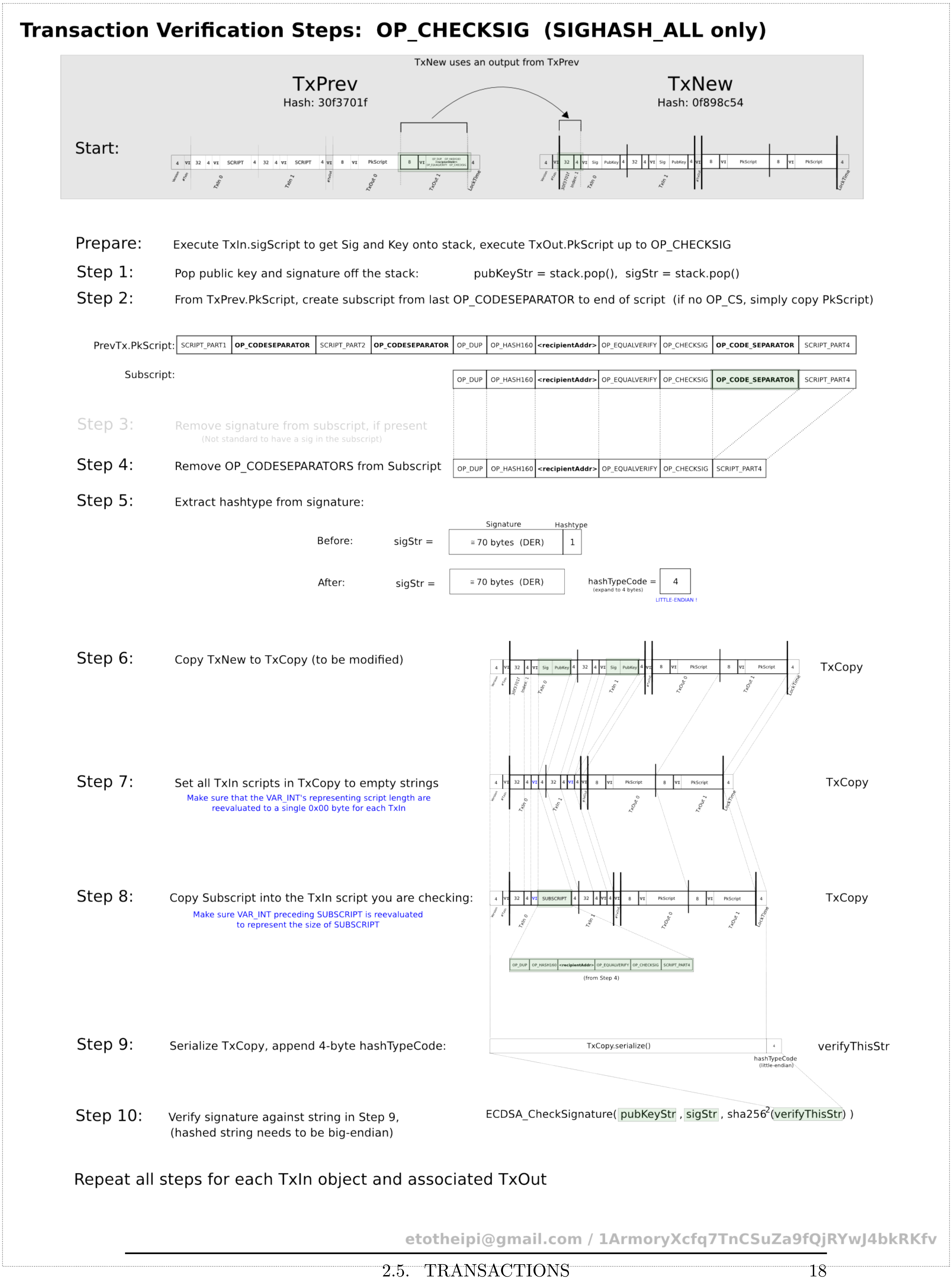
SIGHASH_NONE
This one signs the transaction butwithout the outputs, it could be thought of as "I don't care where the money goes"

SIGHASH_SINGLE
All outputs are removed except the output with the same index as the input that is being signed, then that transaction is signed.

SIGHASH_ANYONECANPAY
Signs the transaction with all the outputs but none of the other inputs. This basically means "The money has to go here, but I don't care if someone else want to fund this transaction also"

More detailed process

On the next page the entire signing process for SIGHASH_ALL is detailed. This is how it is performed in the actual implementation:



Transaction Verification Steps: OP_CHECKSIG (SIGHASH_ALL only)

Prepare: Execute TxIn.sigScript to get Sig and Key onto stack, execute TxOut.PkScript up to OP_CHECKSIG

Step 1: Pop public key and signature off the stack: pubKeyStr = stack.pop(), sigStr = stack.pop()

Step 2: From TxPrev.PkScript, create subscript from last OP_CODESEPARATOR to end of script (if no OP_CS, simply copy PkScript)

PrevTx.PkScript: SCRIPT_PART1 | OP_CODESEPARATOR | SCRIPT_PART2 | OP_CODESEPARATOR | OP_DUP | OP_HASH160 | <recipientAddr> | OP_EQUALVERIFY | OP_CHECKSIG | OP_CODE_SEPARATOR | SCRIPT_PART4
Subscript: OP_DUP | OP_HASH160 | <recipientAddr> | OP_EQUALVERIFY | OP_CHECKSIG | OP_CODE_SEPARATOR | SCRIPT_PART4

Step 3: Remove signature from subscript, if present
(Not standard to have a sig in the subscript)

Step 4: Remove OP_CODESEPARATORS from Subscript

OP_DUP | OP_HASH160 | <recipientAddr> | OP_EQUALVERIFY | OP_CHECKSIG | SCRIPT_PART4

Step 5: Extract hashtype from signature:

Signature
Hashtype
Before: sigStr = ≈ 70 bytes (DER) | 1
After: sigStr = ≈ 70 bytes (DER)
hashTypeCode = 4
(expand to 4 bytes)
LITTLE-ENDIAN !

Step 6: Copy TxNew to TxCopy (to be modified)

Step 7: Set all TxIn scripts in TxCopy to empty strings
Make sure that the VAR_INT's representing script length are reevaluated to a single 0x00 byte for each TxIn

Step 8: Copy Subscript into the TxIn script you are checking:
Make sure VAR_INT preceding SUBSCRIPT is reevaluated to represent the size of SUBSCRIPT

Step 9: Serialize TxCopy, append 4-byte hashTypeCode:

Step 10: Verify signature against string in Step 9, (hashed string needs to be big-endian)

ECDSA_CheckSignature(pubKeyStr , sigStr , sha256^2(verifyThisStr))

Repeat all steps for each TxIn object and associated TxOut

More here? Maybe

2.5.2 Pay to public key hash (P2PKH)

P2PKH is as mentioned the most common form of transaction. This transaction can be thought of as paying to somebodies address. In other words the output of this transaction contains a script where the one who wants to redeem it must prove that they own the private key which created the public key the address is referring too. This is what the script looks like in the output:

```
OP_DUP OP_HASH160 <public key hash> OP_EQUALVERIFY OP_CHECKSIG
```

As mentioned already executing the script in the output by it self doesn't make any sense, especially now when the very first operation `OP_DUP` tries to duplicate the top element in the stack. But at execution the stack will be empty. Anyone wanting to spend this output would construct a transaction with the input script on the following form:

```
<signature> <public key>
```

When the input and output is executed together the process goes like this:

1. First the signature and unhashed public key is pushed to the stack.
2. The public key is duplicated (there are now one signature and two public keys on the stack).
3. The top public key goes through the HASH160 process making it a hashed public key.
4. The hashed public key from the output is pushed on the stack, at this point the stack has the following elements (`<signature>`, `<public key>`, `<hashed public key>`, `<hashed publick key>`).
5. `OP_EQUALVERIFY` checks if the top two elements is equal and verifies the result, What basically has been done is is that the script checks if the public key provided in the input script is equal to the hashed public key given in the output script.
6. At this point the stack has the following elements: (`<signature>`, `<public key>`). `OP_CHECKSIG` pops these from the stack and checks if the signature is valid.

In the early days of bitcoin you payed directly to public keys instead of hashed public key. The reason the hash part was added at all has to do with extra security. If an exploit is found in elliptic curve cryptography that makes it so someone could calculate the private key from a public key then all unspent outputs would be at risk of being stolen. With the hashed public key the actual public key is not revealed until the output is spent, and then it is too late for it to be stolen. This of course requires that everyone uses a different public key for every transaction, which is the standard today.

2.5.3 Pay to script hash (P2SH)

Pay to script hash is slightly newer and a bit more complex to understand. Instead of paying to someones address, you pay to a script. Let's say Alice has partial script S, that can be solved with the partial script K. She can then hash S and get H_S. She then creates transaction with an output containing the following script:

`OP_HASH160` <H_S> `OP_EQUAL`

If Bob wants to redeem this transaction he first has to know the script and how to solve it. This is how the transaction input would look like:

```
K <S>
```

The execution of the combined input output script is not straight forward as most scripts are. This is the process:

1. The K part of the script is executed as usual. This pushes or does whatever it needs to do to make K + S a valid script.
2. The partial script S is pushed to the stack in the form of bytes.
3. This is where the execution takes a strange and not so intuitive path, as you will see. First the script is hashed with the `OP_HASH160` operation.
4. The script hash is form the output is pushed on the stack.
5. OP_EQUAL checks if the top two items on the stack are equal. If they are it means that the script S provided in the input is the correct script.
6. Unique to P2SH, the execution goes back to the original script S that was pushed to the stack in stage 2. And executes it together with whatever K pushed to the stack. This script also has to be valid.

If all stages are passed without warning the redeeming transaction is valid.

P2SH was developed to resolve the difficulties in making complex transaction scripts and make make them as simple as paying to an address. What P2SH basically means is "pay to a script matching this hash, a script that will be presented later when this output is spent"

2.5.4 Timelock and sequence

Transactions have a very useful feature that is widely used in both atomic swaps and payment channels, and that is timelock and sequence. These variables are used for locking a transaction, meaning it can't be included in the blockchain until a certain time has been passed. Timelock is for absolute locks, for example this transaction can't be spent until `10th June 2019`, sequence is for relative timelocks, for example this transaction can't be spent until 10 days has passed since the original output was included in the blockchain.

Both sequence and timestamp can be interpreted as blockheight[1] and time in seconds. For example if the timestamp is equal to or larger than 500 million it is seen as a unix-timestamp if it's less than it is seen as a blockheight.

OP_CHECKTIMELOCKVERIFY

There is an operation in Script called `OP_CHECKTIMELOCKVERITY`, what it does is that it compares the timestamp or blockheight that is on the stack and compares it to the timestamp that is in the transaction trying to spend the output that this op code is part of.

OP_CHECKSEQUENCEVERIFY

`OP_CHECKSEQUENCEVERIFY` is a lot like the previous one except it it deals with relative time.

[1]Blockheight is the number of blocks on the current chain

Let's say you have two transactions transaction T_A and transaction T_B, T_A is included in a block on the blockchain. If T_B tries to spend one of T_A outputs and has that specific input marked witha sequence of 10. It means that T_B can't be included in the block chain until T_A is at least 10 blocks deep in the blockchain.

`OP_CHECKSEQUENCEVERIFY` enforces this in the output. If T_A has `10 OP_CHECKSEQUENCEVERIFY`, then if T_B tries to spend that output it has to be at least 10 blocks deep.

2.5.5 Raw transaction

2.6 Segregated Witness

Segregated witness (sometimes called just segwit) is a relatively recent addition to bitcoin. A witness in cryptography is a proof of knowledge, for example a signature that acts as proof that whoever created the signature knows the private key that created a certain public key. A signature can also act as proof that the signee knew the content of whatever data that was signed.

In bitcoin the script in each input is the witness, and segregated witness means that the witness part of the transaction is separated from the transaction. The witness is still sent with the transaction when broadcasting, but it is no longer part of the transaction structure.

The reason for this change has to do with malleability. If a transaction still has the witness data it can be manipulated in millions of ways, and all these manipulated versions will all be valid. Let's look at an example, If a transaction has the following input script:

```
<signature> <public key>
```

A malicious actor could manipulate the script like this for example:

```
1 OP_DROP <signature> <public key>
```

The script is still valid.[2] Witness manipulation is not a security flaw in terms of monetary loss as the outputs can't be manipulated without breaking the signatures. What is changed however is the txid. The txid is used to identify the transaction, it is the hash of the entire transaction, you could imagine Alice sending money to Bob, Bob manipulates the transaction so that the txid changes, one of the manipulated transactions makes it into a block thus making all other versions, including the original, invalid, Bob got his money but he can now tell Alice that he didn't. Alice tries to find her transaction in the blockchain with the txid she had but it can't be found.

Segregated witness was activated via a soft fork on `21st July 2017`, it's purpose is to make the Bitcoin network ready for lightning network usage. Segregated witness is not a necessity for payment channels and lightning network. But it does make it significantly safer. A great example is the funding of payment channels. The funding transaction requires the signature of both parties to be spent, to prevent a malicious actor from broadcasting the funding transaction and locking both participants money eternally both signs the initial commitment transaction. The commitment transaction of course use the id of txid of the funding transaction as reference. With malleable transactions a malicious actor could modify the funding transaction and broadcast it. The commitment transaction would be useless as it's referenced txid doesn't exist.

As mentioned segwit separates the witness and transaction. This makes it so the witness is no longer part of the txid, making it effectively immutable. The witness data is stored

[2] A transaction with that input script would be rejected in current implementation, but it shows the concept of mallebility pretty well

separately to the blockchain and it's byte-size is not counted towards the block size, making it so more transactions could fit in a single block.

2.7 On-chain Atomic swaps

Onchain atomic swaps is a method where two parties can exchange cryptocurrencies between different chains in an atomic way, in this case it means that none of the parties can cheat the other and no matter what state in the exchange-process the swap is either completed or reversed entirely, to the pre-exchange state. The onchain distinction comes from the fact that this type of atomic swap occurs entirely on the chains (There is no offchain transaction in the process).

There are several ways atomic swaps can be performed, scripts, signatures and time locks can take several different combinations in transactions and still retain atomicity. The method that will be described in this report will be a P2SH, pay to contract type. This swap is done with the help of time locked contracts constructed with scripts.

2.7.1 The contracts

A swap contract contains certain components. The initial component is the secret x that is generated by the initial contract creator. The secret is not included in the initial contract however, this is what is included:

- Byte string h - Hash of secret x
- Integer T - Contract expiration time as timestamp
- Address B - Redeemer address
- Address A - Refund address

The contract uses branching provided by the operations `OP_IF`, `OP_ELSE` and `OP_ENDIF`. The branch that is taken during execution is decided by whoever is trying to spend the output. One branch pays to the redeemer address, for it to be valid the secret has to be revealed. The other branch pays to the refund address, but it can only be valid if the current time is equal to or greater than the timestamp T.

Counter party contract

The contract described above is the one constructed by and broadcast by whoever wants to initialize the exchange. A very similar contract has to be constructed and broadcast by the counter party, this contract however contains a few changes. The only variable that remains unchanged is h. T has to be be a value between the current time and T from the old contract, denoted $T/2$. Redeemer and refund addresses has to be changed so that the redeemer is whoever constructed the initial contract and the refund address is yourself.

2.7.2 The process

Imagine Alice and Bob wants to swap cryptocurrencies. Alice has Bitcoin and wants Litecoin, Bob vice versa. Alice is the initiator in this case. The process would be the following:

1. Alice generates a new secret and then constructs a contract from the variables needed as stated above.
2. She broadcasts a p2sh transaction (to the Bitcoin blockchain) that pays to the constructed contract, called the contract transaction. Alice then sends the contract and the contract transaction txid to Bob.

3. Bob fetches the transaction from the Bitcoin blockchain and validates all the variables, as well as the contract to contract hash etc...
4. If all seems to be in order, Bob can construct his own contract using the same secret hash h as Alice did in her contract. This time however the timelock is set to $T/2$. He then broadcasts a transaction paying to the contract on the Litecoin network. Just as Alice did, Bob sends the contract and txid to Alice.
5. Alice validates all the data the same way that Bob did in step 3.
6. If all seems to be in order. Alice could claim the Litecoins from Bobs contract transaction by making a transaction spending the output. The input that validates the output script must contain the secret x to be valid.
7. Meanwhile Bob monitors the Litecoin chain for someone spending his contract transaction. If somebody does it means that they know the secret x. Any information in the blockchain can be extracted, Bob needs to know x to spend Alice's contract transaction. If somebodies spends his contract transaction he can extract x from that and spend the Bitcoin side transaction.

Visualizer
The next page contains a diagram visualizing the process described above. T is set to 48 hour after initialization in the example, thus T/2 would be 24 hours.

2.7.3 Atomicity

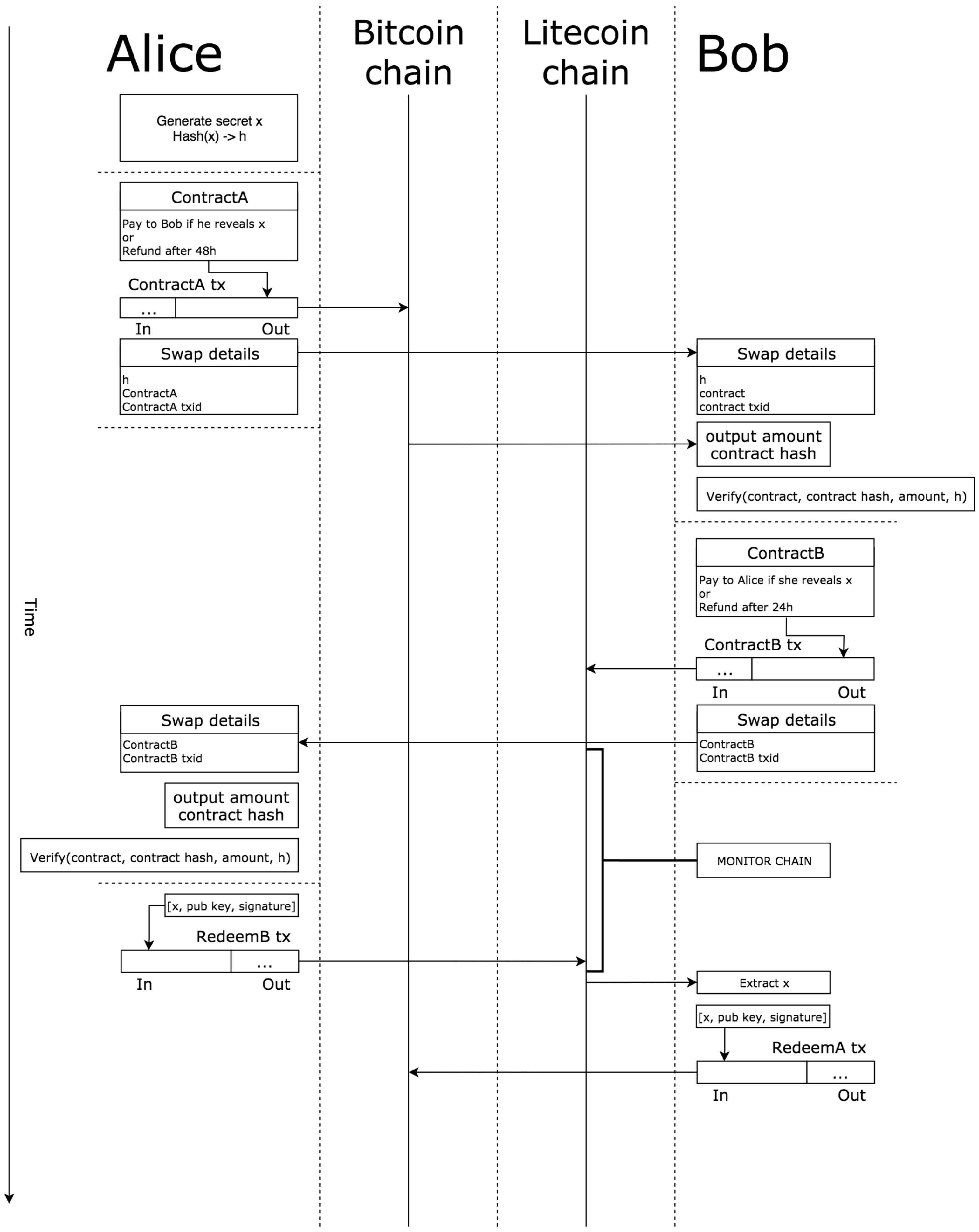
Easiest way to show the atomicity of the swap is to show what happens if one party stops cooperating during any step in the process. The two trivial cases are before any step in the process, then no transaction has been made and the state did not change at all. If one becomes uncooperative at the end of the process the state change has already been completed and it doesn't matter what anyone does. The non-trivial cases is when someone becomes uncooperative mid process.

The first case we will look at is if Bob stops responding after step 2. Meaning he never broadcasts his side of the contract. Then Alice simply never reveals x, Bob needs x to claim the Bitcoins. Alice then just waits on till timelock T has passed and refunds the transaction. After the refund the state has been reset to the initial state.

Can Alice wait to the last second before she claims the Litecoins?

Yes, but remember that Bob's contract expires before Alice's contract does: $T > T/2$, Even if Alice waits to the very last second, Bob will still have an equal amount of time to claim Alice's Bitcoins after that.

The second case is if Alice becomes unresponsive after Bob has broadcast his side of the contract. Bob can't do anything until he knows x or the timelock ($T/2$) of his transaction expires. If Alice does nothing Bob will never know x and thus has to wait to refund his transaction. Presumably Alice does the same after timelock T expires. The state has been reset to the initial one.


Time
Alice
Bitcoin chain
Litecoin chain
Bob
Generate secret x
Hash(x) -> h
ContractA
Pay to Bob if he reveals x
or
Refund after 48h
ContractA tx
...
In
Out
Swap details
h
ContractA
ContractA txid
Swap details
h
contract
contract txid
output amount
contract hash
Verify(contract, contract hash, amount, h)
ContractB
Pay to Alice if she reveals x
or
Refund after 24h
ContractB tx
...
In
Out
Swap details
ContractB
ContractB txid
Swap details
ContractB
ContractB txid
output amount
contract hash
Verify(contract, contract hash, amount, h)
MONITOR CHAIN
[x, pub key, signature]
RedeemB tx
...
In
Out
Extract x
[x, pub key, signature]
RedeemA tx
...
In
Out

2.8.1 How the swap could fail

The atomicity of the swap depends on each party acting rationally and not doing any mistakes that compromises the process. For example if Alice broadcasts her side of the contract and then reveals x to Bob thorough some alternative communication channel then Bob could claim the Bitcoins without doing his part of the deal.

Malformed contracts could also lead to money being stolen. This is why the validation steps are very important. For example Alice could maliciously set her contract to expire earlier than Bobs if Bob is not careful.

Actors in the scenario doing nothing also compromises the swap. If Alice waits til after $T/2$ expires before attempting to claim the Litecoins she is risking out on losing both the currencies. Bob could monitor the transaction pool, seeing what x was, then sneak in his own refund transaction reclaiming his Litecoins (This is possible as the refund branch of the contract now is unlocked). Then also claiming the Bitcoins before T expires.

2.9 Payment channels

A payment channel is a channel where transactions can be exchanged trustlessly on channels other than directly on the blockchain. This is what is sually refered to as off-chain transactions. In their most naive form a payment channel can simply be Alice and Bob exchanging promises of future transactions later. This however is not trustless and any of the parties could later withdraw from the promise without punishment (other than maybe loss of friendship).

To build a trustless payment channel there are several ways to go about as Script is quite versatile. The method that will be covered here uses a type of channel **funding transaction that requires the signature from both parties to spend**, and the channel balance is updated via special commitment transactions that spend the funding output. Each party in the channel has their own version of the commitment transaction. Whenever a commitment transaction is broadcast to the blockchain the channel is closed as the funding transaction can't be spent twice. To update the balance in the channel both parties create new commitment transactions and signs the other parties commitment transaction. Let's take a look at a naive example:

The naive payment channel

Imagine Alice and Bob wants to open a payment channel between eachother. They create a funding transaction and the initial commitment transactions. Let's say they both funded the channel with 1 Bitcoin each. The initial commitment transactions would then have one output paying 1 Bitcoin to Alice and one output paying 1 Bitcoin to Bob, let's call this commit (**C0**). The funding transaction is braodcast to the blockchain.

A bit later Alice buys one funny hat from Bob for 0.1 Bitcoins. To complete the transaction they create two new commitment transaction that pays 0.9 Bitcoin to Alice and 1.1 Bitcoin to Bob and signs them for each other, let's call this new commitment (**C1**). Any number of transactions could be exhanged this way. The channel could be closed by any one broadcasting their commitment transaction. This naive implementation of a payment channel has a fatal flaw however.

A payment channel needs some safety measures to be safe from malicious actors. The above example lacks a mechanism for preventing old commitment transactions from making their way into the blockchain and still paying the malicious party.. For example after the initial funny hat transaction above (**C1**) Alice could just broadcast the initial commit transaction (**C0**) and reclaim the 0.1 Bitcoins she spent. Bob would have no way of preventing this in this naive implementation.

Transaction flow diagrams

The transactions that are involved in making payment channels grows larger the more features that are added. To make it easier to understand, diagrams are used together with the descriptions. In Figure 2.4 a basic overview of the diagram-realm transaction is shown. Lines leading from outputs to inputs means that the connected input are spending that output. Note that one output can be connected to several inputs but each input can only be connected to one output, of course an output can only be spent once, this will be clarified later. A **?** in the broadcaster field indicates either that the sender is unknown or that who sends it is irrelevant.

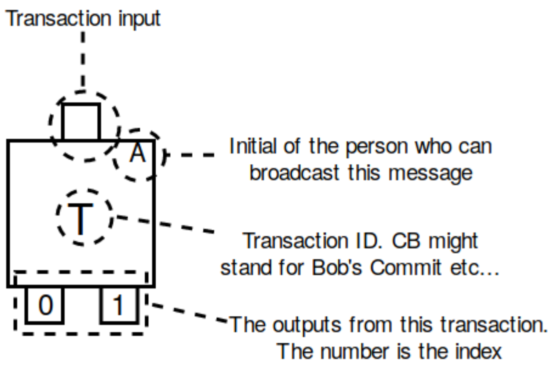

Transaction input
A
Initial of the person who can broadcast this message
T
Transaction ID. CB might stand for Bob's Commit etc...
0
1
The outputs from this transaction. The number is the index

Figure 2.4: Different parts of a transaction as they appear in diagrams.

Payment channel with breach remedy

To prevent old transactions from being sent to the blockchain a new mechanism needs to be devised. This can be done via something called revocable delivery and breach remedy (Some times called just revocation). Instead of both parties holding the same commit. They each get an individual one that differs in it's outputs.

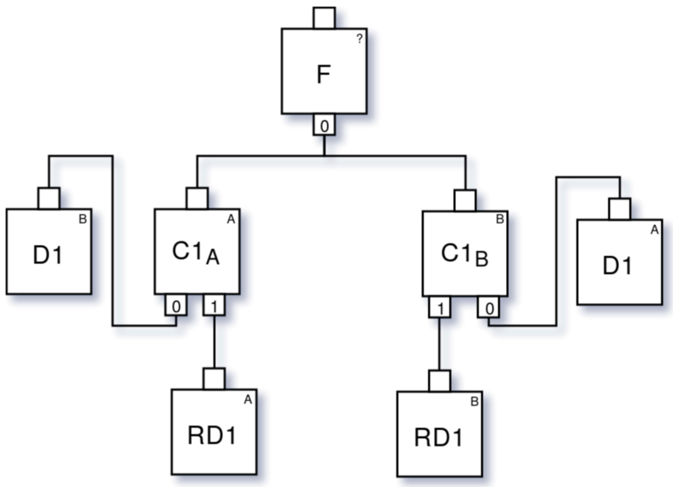


Figure 2.5: Payment channel, with commits using revocable delivery mechanism

Figure 2.5 shows how the payment channel appears after the first commitment transactions has been made. Let's say the channel is between Alice (A) and Bob (B) and the balance in the channel is 0.5 BTC for Alice and 0.5 BTC for Bob.

Lets' take a look at the left side of this diagram. $C1_A$ is the commitment transaction on Alice's side of the channel. As with all commitment transactions it spends the funding transaction. It has two outputs. Output 0 sends 0.5 BTC to Bob completely unencumbered. Output 1 is a bit more complicated however. It pays into a revocable delivery contract worth 0.5 BTC.[3] The contract is constructed with a relative time lock that allows Alice to claim her share of the money after x amount of time (relative to the broadcast time of $C1_A$, opften described in terms of blocks) or pays to whomever can sign using the pre-generated revocation keys (See section 2.3.6)

D1 is the transaction Bob uses to claim his money in case Alice decides to broadcast her commitment transaction ($C1_A$).

RD1 is the transaction that Alice uses to claim her money. The transaction has a relative timelock on it that makes it so it can't be broadcast until x blocks has passed since $C1_A$ was included in the blockchain. **From now on the relative timelock will be 100 blocks in all the following examples**.

The right side is identical, only that the RD1 and D1 relationship is reversed, meaning that Alice caan claim her money directly and Bob has to wait.

If Alice wants to send 0.1 BTC to Bob they need to update the channel with new commitment transactions and then invalidate the old transaction. The transaction diagram will now look like figure 2.6

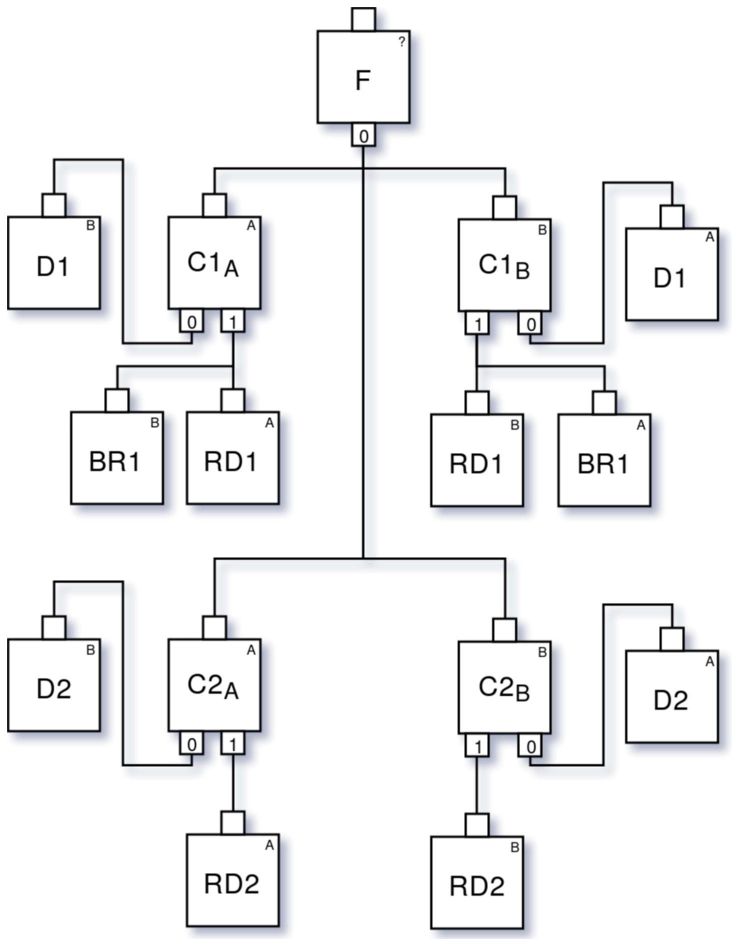


Figure 2.6: Updated payment channel, with breach remedies for old commitment transactions

[3]The contract is a script, and it's payed to via P2SH

Figure 2.6 shows the state of the channel after the new commitment transactions have been finalized. $C2_A$ is a lot like the previous commitment transaction with a few changes. The primary change is of course the channel balance, output 0 now has 0.6 BTC as value and output 1 has 0.4 BTC (Because Alice sent 0.1 BTC to Bob).

Another important update to the channel is the breach remedy. BR1 is the breach remedy for the first commitment transaction, it's purpose is to prevent old commitment transactions from being broadcast. The breach remedy can be created after Alice has revealed the secret private key she used when creating the revocation key. Of course to prevent a new false breach remedy to be created for $C2_A$ Alice uses a new revocation key for all new commitment transactions. If Alice tries to take back her money by broadcasting an old commitment transaction Bob can prevent it by broadcasting the breach remedy, this is possible because of the timelock on Alice's claim being time-locked.

These are the steps taken whenever someone wants to send money in the channel (Under the assumption that commitment n-1 (C_{n-1}) was the latest commitment transaction in the channel), Alice is assumed to be the sender:

1. Both parties create new revocation public keys.
2. Alice creates the commitment on Bob's side (Cn_b) using the revocation key from the previous step, signs it and sends it to Bob.
3. Bob creates the commitment on Alice's side (Cn_a) using the revocation key from the previous step, signs it and sends it to Alice.
4. Each party generates their repective Delivery and Revocable delivery transactions (Dn and BRn)
5. Both parties reveal the secret used to create the revocation keys for the previous commitment transactions (C_{n-1})
6. From these revocation keys each party constructs the breach remedy (BR_{n-1}) for the previous commitment on the other persons side.

These steps can be repeated however many times until the channel is closed.

Naive payment network

The current form of our payment channel works very well as long as it is between just two parties. A very helpful feature would be to be able to send money to anyone without having a direct channel between you and that person, in other word the transaction jumps between any number of other channels before reaching it's destination. If we naively use the channel we have constructed in the previous section as is to send money over multiple jumps we would run into the following problem:

Imagine there being 3 people: Alice (A), Bob (B) and Cecil (C). Alice and Bob have a channel between each other and Bob and Cecil have a channel between each other. Making the following graph: A - B - C. Alice wants to send money to Cecil but they have no channel between each other. To solve this Alice asks Bob to promise to send x amount of money to Cecil if Alice sends x amount of money to Bob. In the perfect world Bob would be honest and this system of payment networking would work. In the real world however having to trust every node between you and the destination would cause problems with malicious nodes. There is nothing stopping Bob from taking Alice's money and never sending anything to Cecil.

The channel has to be extended further to allow multi-hop payments and still retain it's breach remedy feature. This is where Hashed Time Locked Contracts come into the picture.

Payment channels with Hashed Time Locked Contracts

A new mechanism needs to be added to our payment channel to enforce that multi-hop transactions actually make it all the way to their destination while at the same time still retaining the breach remedy mechanism etc...

This could be done by adding a third output to the commitment transactions. This new output pays to a hashed time locked contract (**HTLC**). Before when a payment was made over a payment channel the amount was updated for each output representing a user in the channel. With the HTLC-output the amount sent is represented by the the value of the HTLC-output.

A channel could be thought of as having three states: regular, unsettled, and settled. The regular state is the one described in previous sections, it's a channel with one set of active commits, the commit will not have any HTLC-outputs. A channel where the latest commits has a HTLC-output is considered unsettled. The unsettled state should only exists while there is uncertainty about whether the multi-hop transaction completed or not. You could say that **the HTLC-output only exists as a contingency**, if both parties cooperate the channel can return to regular state with the channel balances updated accordingly, if the parties cannot agree or for some reason will not cooperate the channel should be close as soon as possible, as to why we will get to that. A channel reaches a settled state when both parties decides to close the channel, it is done by creating one last pair of commitment transactions but without any revocable deliveries. After a channel is settled no further transactions could be exchanged in it safely as the mechanism for preventing old transactions from being broadcast has been removed, instead the commitment transaction should be broadcast and the channel properly closed.

To understand the HTLC-output you first need to understand the general mechanism that is used to ensure that multi-hop transactions work.

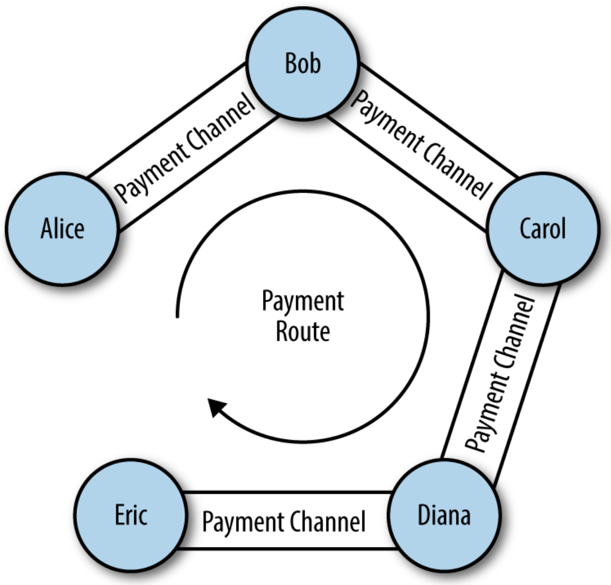


Figure 2.7: An overview of a network formed by payment channels

The principle used for multi-hop transactions is a lot like the one used for onchain atomic swaps. It relies on revealing a secret pre-image[4] of a hash within a certain time limit. Using the diagram in figure 2.7 as example. Alice wants to send money to Eric. But they have no direct channel between each other. To make this work Alice asks Eric for a hash of a secret, let's call this $H = SHA256(R)$. For now Eric holds on to R and only sends H to Alice.

For the money to reach Eric, Alice needs to go via Bob. To do this Alice requests from Bob that they make a new commitment transaction with one HTLC output. The amount in the output should be equal to to whatever amount Alice want to send to Eric. The promise here is that Bob will get this money only if he can prove that he has the pre-image to H. Bob does the same with Carol. This continues until an updated commitment reaches Eric. Eric can then send the pre-image R backwards through the chain until it reaches Alice.

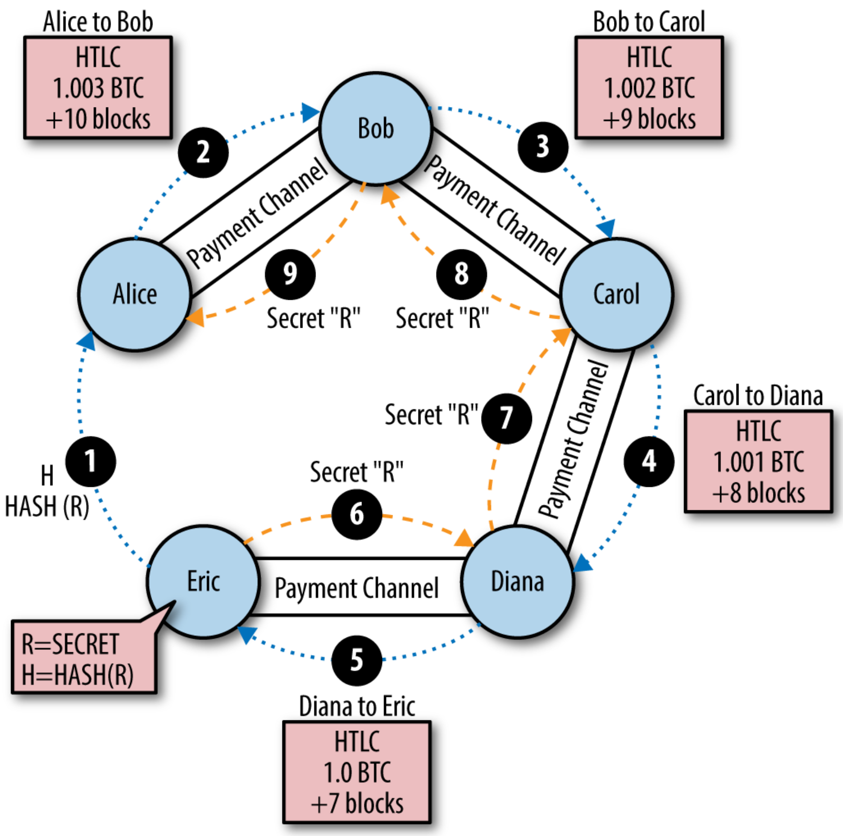


Figure 2.8: All the steps taken in the payment network

Earlier it was mentioned that the HTLC output only existed as a contingency, when Bob can prove to Alice that he knows the pre-image R, and can claim the money in the htlc-output by closing the channel, Alice has no reason to not update the channel with correct balances in Bob's favor. Thus the channel returns to a regular state with no htlc-outputs and all previous commits are revoked. This is done for all channels between Alice and Eric. If a party will not co-operate in the chain the channels to that person will have to be closed, this is the only case where the htlc-outputs are actually spent. There is no risk of monetary loss, closing channels is generally something that should be avoided if not absolutely necessary.

HTLC in more detail and second-level HTLCs

The HTLC script is a bit different depending on what side you are on in the channel, in other words the HTLC in the commitment of the sender is different to the one on the

[4]See glossary

receiver end. What they both have in common however is the three possible outcomes. Let's call these paths **HTLC timeout**, **HTLC execution** and **HTLC breach remedy**.

HTLC execution can be considered to be the main path and the one that is taken if everything goes as planned. Let's take a closer look at what happens in the channel between Alice and Bob in figure 2.8. Bob knows the secret R because Carol sent it to him. Under normal circumstances Bob would prove to Alice that he knows the secret R, Alice would accept and the channel would be updated to reflect the transaction taking place. But for the sake of example let's say that Alice is uncooperative and does not want to give up the money she just sent to Eric. Bob should then close the channel with Alice and claim his money in the HTLC output by providing a input containing the pre-image (R), and signatures from both Alice and Bob. The reason for both signatures being needed has to do with second-level HTLC, but we will return to that in a moment. The signatures are provided prior to the new commitment transaction.

HTLC timeout is the path taken in the case where Bob never received the pre-image (R) from Carol. In this case the money should be refunded to Alice. Under normal circumstance Bob sees that Alice can claim the money and thus updates the channel to reflect the refund. In the case where Bob refuses to cooperate Alice can close the channel after the timeout has passed. This time no pre-mage is needed, only the signatures from both channel parties. As with the HTLC execution the transaction spending the HTLC output must pay into a second-level HTLC.

HTLC breach remedy does not pay the money into a second-level HTLC instead it pays the money directly to the party that did not send an old commitment transaction. The breach remedy path is redeemed by providing a correct signature with the revocation key.

The second-level HTLC is a necessary second part of the HTLC output. Whenever the HTLC-output is spent (except in cases of revocations) it has to lead into a second-level HTLC script, this is enforced by the HTLC script requiring the signature from both parties to be spent. The second-level HTLC script is fairly basic, it has two outcomes. Either the spender has to wait x amount of blocks (in relative time) before spending, or it can be revoked immidietly if a signature with the revocation keys is provided. It's usefulness can be demonstrated with an example:

Take the example where a transaction from Alice to Eric passed through a channel between Alice and Bob. Let's say a couple of days passed and Alice and Bob has exchanged other transactions since. Without second level HTLC Bob could broadcast the old commitment transaction and try to claim the HTLC-output money by providing the pre-image. With second-level HTLC Bob needs to wait a relative time before the money properly becomes his. If the old commit was properly revoked Alice could revoke Bob's attempt at claiming the HTLC-ouput. Same holds true the other way around if Alice tries to claim an old HTLC output by waiting for the timeout.

Figure 2.9 is the equivalent transaction flow diagram for a channel where the third commit contains a HTLC-output. To keep the diagram from exploding completely all the previous and following commitment transactions has been excluded, as well as Bob's side of the commitment tree.

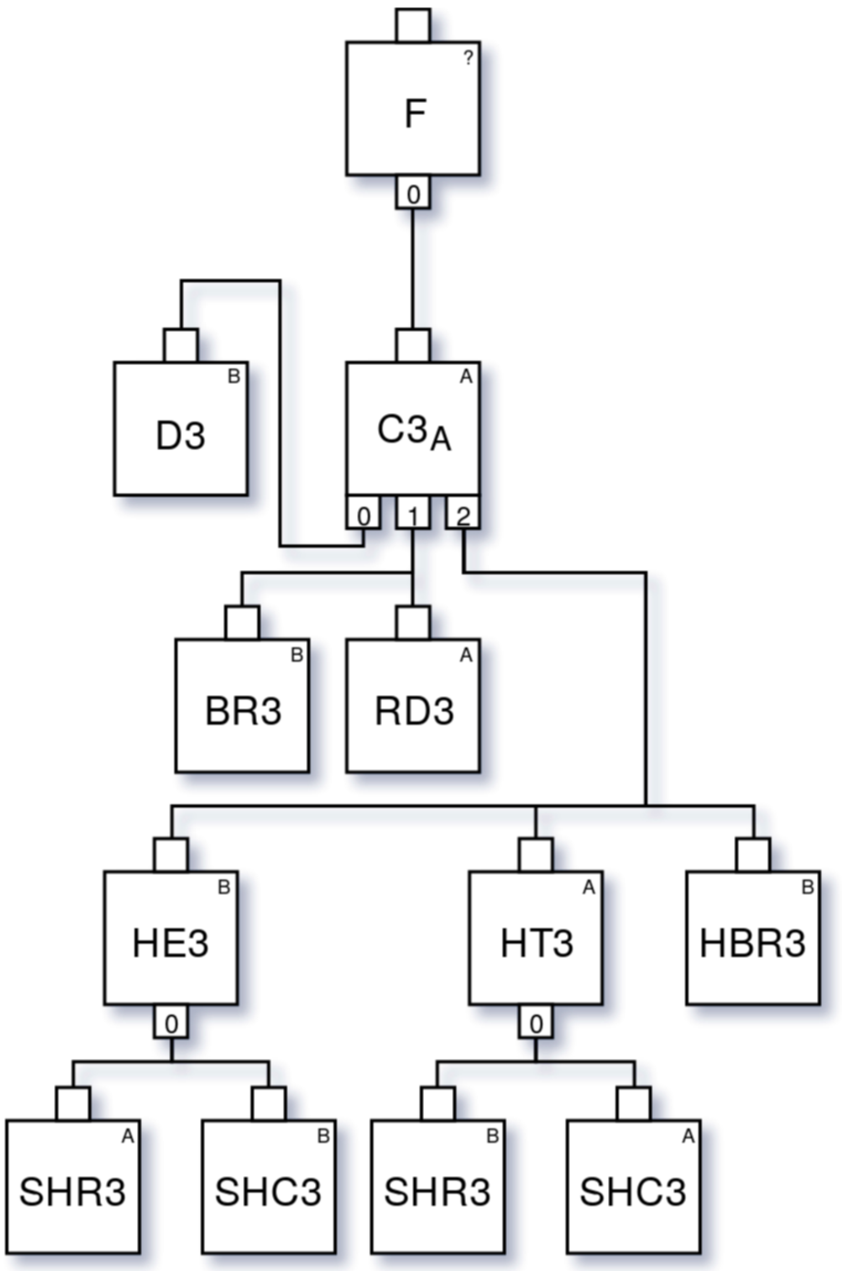


Figure 2.9: The transaction diagram showing the htlc-output this time

2.10 Off-chain atomic swaps

Chapter 3

Implementation

Chapter 4

Comparison

Chapter 5

Conclusion & Discussion

Chapter 6

Future research

Bibliography

[1] Genesis block. https://www.blockchain.com/sv/btc/block-height/0.

[2] On scaling decentralized blockchains. https://www.comp.nus.edu.sg/~prateeks/papers/Bitcoin-scaling.pdf.

[3] The original description of atomic swaps on bitcointalk. https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

[4] A post by satoshi on bitcointalk. https://bitcointalk.org/index.php?topic=234.msg1976#msg1976T.

[5] A survey of bitcoin transaction types.

[6] Text dump of nakamotos last mail. https://pastebin.com/syrmi3ET.

[7] Andrea M. Antonopoulos. *Mastering Bitcoin: programming the open blockchain.* OReilly, 2 edition, 2017.

[8] Zoe Bernard. Everything you need to know about bitcoin, its mysterious origins, and the many alleged identities of its creator. *Business Insider*, Nov 2018.

[9] Adrianne Jeffries. Four years and $100 million later, bitcoin's mysterious creator remains anonymous. *The verge*, May 2013.

[10] Lightningnetwork. lightningnetwork/lnd, Jan 2019.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[12] Certicom Research. Sec 1: Elliptic curve cryptography, May 2009.

[13] Certicom Research. Sec 2: Recommended elliptic curve domain parameters, January 2010.