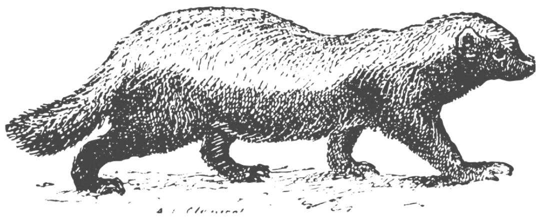
A formal programming model for Bitcoin transactions

a master's thesis
by
Bjarne Magnussen
to
The Department of Mathematics and Computer Science

For the degree of
Master of Science
in the subject of
Applied mathematics

University of Southern Denmark
Odense, Denmark
June 2016

Supervisor: Fabrizio Montesi
Co-superadvisor: Luís Cruz-Filipe



THIS THESIS WAS TIMESTAMPED on the 1st of June 2016 using the Bitcoin blockchain. A verification of it can be done online at proofofexistence.com. The above illustration depicts a honey badger, which has become a beacon for those who believe that digital currency is destined to take over the financial world. This document was typeset using the XƎLaTeX typesetting system. The font used for the body text was set with Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. Mathematics was set with *XITS Math*. A template, which can be used to format a thesis with this look and feel, has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.

— Money …? in a voice that rustled.
— Paper, yes.
— And we'd never seen it. Paper money.
— We never saw paper money till we came east.
— It looked so strange the first time we saw it. Lifeless.
— You couldn't believe it was worth a thing.

William Gaddis, *J R*

Abstract

BITCOIN IS A DECENTRALIZED peer-to-peer electronic currency system. As its basis serves a composition of different areas in computer science, such as distributed systems and cryptography. The first part of this thesis describes the mechanisms behind Bitcoin in detail.

Transactions in Bitcoin contain a list of instructions describing how the transferred bitcoins can be spend again by e.g. the recipient. This list of instructions has an underlying scripting system, which uses a programming language called Script. It allows not only to send bitcoins from a sender A to a receiver B, but to express complex conditions for spending. Script is a very low-level language based on a stack machine without support for abstractions. The second part of this thesis contains the main focus of my research and regards this scripting system. A formal model was created to reason about the syntax and semantics of Script. This allowed to identify potential errors that can occur in the programming and execution of scripts. A high-level language was then developed, which introduces abstractions such as variables and a type system to prevent erroneous scripts from being compiled. Finally the applicability of the newly developed language was investigated using existing scenarios of scripts that resemble complex contracts in transactions.

Contents

Tables

Preface

Before you lies the thesis *"A formal programming model for Bitcoin transactions"*. I was engaged in researching and writing this thesis in the months from June 2015 to June 2016. I first heard about Bitcoin in 2011, when reading about it online. I was immediately fascinated by its technical principles. Incidentally I hope that this thesis can convince you that technologically Bitcoin is deep, novel, fascinating, and based on sound principles. Bitcoin has opened up a new world that has just started to be explored.

Being engaged with Bitcoin, it is nearly impossible to not also be influenced by its non-technical sides. Especially in its early days, Bitcoin seemed to have polarized opinions. Personally though, it has widened my comprehension of what money is, both economically, politically, and psychologically.

I am therefore grateful of having been able to write my thesis on Bitcoin under supervision by Fabrizio Montesi and Luís Cruz-Filipe, who helped me to formulate the aim of this research, and were always available and willing to answer my questions. I would like to thank them for their guidance and support during this whole process.

The research was difficult, and at times frustrating. Bitcoin has emerged out of the cypherpunk community, and information can sometimes be cluttered in different forums or mailing lists on the internet. Especially the books "Mastering Bitcoin", by Andreas Antonopoulos and "Bitcoin and Cryptocurrency Technologies", by Arvind Narayanan were of great preliminary value to me. However, documentation is impossible to comprise all of Bitcoin, and therefore I had sometimes to investigate the source code on specific details. But it has been with great pleasure and passion that I was able to conduct this research on Bitcoin.

Odense, May 2016 *Bjarne Magnussen*

Glossary

address A bitcoin address looks like 1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV. It consists of a string of letters and numbers starting with a "1" (number one). Just like you ask others to send an email to your email address, you would ask others to send you bitcoin to your bitcoin address.

bip Bitcoin Improvement Proposals. A set of proposals that members of the bitcoin community have submitted to improve bitcoin.

bitcoin The name of the currency unit, the network, and the software.

block A grouping of transactions, marked with a timestamp, and a fingerprint of the previous block. The block header is hashed to produce a proof of work, thereby validating the transactions. Valid blocks are added to the main blockchain by network consensus.

blockchain A list of validated blocks, each linking to its predecessor all the way to the genesis block.

confirmations Once a transaction is included in a block, it has one confirmation. As soon as another block is mined on the same blockchain, the transaction has two confirmations, and so on. Six or more confirmations is considered sufficient proof that a transaction cannot be reversed.

difficulty A network-wide setting that controls how much computation is required to produce a proof of work.

difficulty retargeting A network-wide recalculation of the difficulty that occurs once every 2.106 blocks and considers the hashing power of the previous 2.106 blocks.

difficulty target A difficulty at which all the computation in the network will find blocks approximately every 10 minutes.

fees The sender of a transaction often includes a fee to the network for processing the requested transaction. Most transactions require a minimum fee of 0,5 mBTC.

genesis block The first block in the blockchain, used to initialize the cryptocurrency.

input An input in a transaction contains four fields: an outpoint, a unlocking script, and a sequence number. The outpoint references a previous output, and the unlocking script allows spending it.

lock script The script that is used in each output and "locks" the amount of bitcoins to the conditions specified inside it.

miner A network node that finds valid proof of work for new blocks, by repeated hashing.

output An output in a transaction contains two fields: a value field for transferring zero or more satoshis and a lock script for indicating what conditions must be fulfilled for those satoshis to be further spent.

Proof-Of-Work A piece of data that requires significant computation to find. In bitcoin, miners must find a numeric solution to the SHA256 algorithm that meets a network-wide target, the difficulty target.

reward An amount included in each new block as a reward by the network to the miner who found the Proof-Of-Work solution. It is currently 25BTC per block.

satoshi Denominations of Bitcoin value, usually measured in fractions of a bitcoin but sometimes measured in multiples of a satoshi. One bitcoin equals 100.000.000 satoshis.

scriptPubKey See lock script.

scriptSig See unlocking script.

transaction In simple terms, a transfer of bitcoins from one address to another. More precisely, a transaction is a signed data structure expressing a transfer of value. Transactions are transmitted over the bitcoin network, collected by miners, and included into blocks, made permanent on the blockchain.

unlocking script The script that is used in inputs and "unlocks" the referenced output by satisfying its corresponding lock script.

The one thing that's missing, but that will soon be developed, is a reliable e-cash. A method where buying on the Internet you can transfer funds from A to B, without A knowing B or B knowing A.

Milton Friedman

1 Introduction

BITCOIN IS AN ONLINE PAYMENT SYSTEM released as open-source software in 2009. In the system, units of currency called bitcoins are used to store and transmit value among participants in the bitcoin network. The network is peer-to-peer: transactions are verified by the participating nodes and recorded in a public distributed ledger called the blockchain, of which every peer has a copy.

Bitcoin combines different areas of research and technologies, such as but not limited to public/private key cryptography, distributed peer-to-peer networks and game theory. Unlike traditional currencies, bitcoins are entirely virtual. There are no physical coins or even digital coins per se. The coins are implied in transactions that transfer value from sender to recipient. Transactions contain a list of instructions that describe how the bitcoins being transferred can be spent again by, e.g., their new owners. The list of instructions in a transaction is programmable with a programming language called Script, enabling the execution of complex transactions. For example, a multisignature transaction may specify that three parties control an account jointly, but any combination of two can together spend received funds. This can be applied, e.g., to add an escrow functionality to the otherwise irreversible Bitcoin transactions. Andrychowicz et al. showed how Bitcoin transactions can also be programmed and implemented to allow for Secure Multiparty Computation [2]. The focus of this thesis regards the way in which Bitcoin transactions can be conditionally programmed using this programming language called Script.

Part 1 of this thesis regards the underlying mechanisms on how Bitcoin works. It describes and combines all the technologies that make up Bitcoin. From part 1, mainly Chapter 4 about the transaction structure is of particular importance for the aim of this thesis, while the rest of part 1 can be read to obtain a better general understanding on how Bitcoin works. Part 2 contains the predominant piece of work of this thesis.

Motivation

Even though there has already been done considerable research on the security, stability and scalability of the Bitcoin network itself, there is still only considerably little knowledge about the Script programming language and its implications on Bitcoin transactions. Understanding this aspect is challenging, since Script is a very low-level language based on a stack machine without support for high-level programming abstractions, e.g., iteration control like for-loops.

Sophisticated applications can nevertheless be implemented in Bitcoin, of which Script constitute the cornerstone. Script is an important and integral part of the Bitcoin system, which needs further investigation.

Aim

In this master thesis, we want to investigate the theoretical underpinnings of the Bitcoin blockchain technology, with particular focus on the Script programming language.

Contributions

The key development will be the creation of a formal model to reason about the syntax and semantics of Script programs. I will use the model to identify some potential errors that can occur in the programming and execution of Script programs, which could otherwise lead to permantently unspendable transactions. Then I will develop a high-level language featuring constructs that prevent such errors, equipped with a compiler towards Script. Finally, I will evaluate my approach against existing applications implemented in Script, showing the applicability of the newly developed language, and also set it in perspective to future compatibility of the Bitcoin system.

Conclusion

The details regarding the Bitcoin transaction structure and spending mechanisms are complex, complicated and in some cases cumbersome. Using a mathematical formalized model therefore allows to formally reason about them.

The high-level programming language developed in this thesis is called NextScript and uses a type system to prevent execution of erroneous programs. Further abstraction was introduced with variables, which Script lacks. Variables allow to program and analyze code in a more straight forward manner. But specifically with Script in mind, they also rigorously and predictably define how the bitcoins in a transaction can be spent again. Compact Script programs are favourable. But for the sake of having a tightly structured compiler, its limitation lies in the production of optimized Script code.

Lastly, there is a lot of development happening in Bitcoin. Since it is an open-source project, in principle anyone can contribute to it. Some ideas are discarded long before even their implications are considered, but others are actively developed and employed into Bitcoin. It stands left to be seen if the use of complex Script programs in Bitcoin will ever seriously take off. But evidence point in that direction for the future.

Three eras of currency
Commodity based, e.g. Gold
Politically based, e.g. Dollar
Math based, e.g. Bitcoin

Chris Dixon

2 Mathematical prerequisites

As a prerequisite for understanding how cryptocurrencies and Bitcoin in particular work, some basic properties and cryptographic methods will be presented in this chapter.

2.1 Hash functions

Bitcoin extensively uses cryptographic hash functions. A hash function is a mathematical function with the following properties:

- the input can be a string of any size,
- the output is of fixed size, typically denoted in the number of bits, and
- it is efficiently computable, typically in linear time.

Some of the more general and important properties for hash functions used in Bitcoin will be described with more details now. Additional and more specific properties for Bitcoin will be discussed later on, when they will be useful for the understanding of the Bitcoin protocol.

Properties of hash functions

Collisions are said to occur when two distinct inputs produce the same hash as output. By a simple counting argument, it is obvious that collisions must exist in every hash function, since the input space to a hash function contains strings of all lengths, but the output space is of specific length.

One can therefore always find a collision by calculating the hash for a number of distinct inputs corresponding to the number of total possible outputs, and then one more. But this brute force method is infeasible, since the calculation required grows exponentially with the size of the output space for the hash function. Although in practice, finding a collision using the *birthday attack* [20] only requires examining the square root of the number of possible outputs; this is still an astronomically large number for common hash functions with 256-bit outputs and therefore still infeasible.

This does not guarantee that there never exists an easy method to find collisions for a hash function with a sufficiently large output space though. The hash function $H(x) = x \bmod 2^{256}$ for example has an output space of 256 bits. But for any input a collision can easily be calculated by just adding 2^{256} to it.

Hash functions, for which no such easy methods are known to exist, are considered *collision-resistant*. However, for many hash functions once thought to be collision-resistant, techniques have been found that are more efficient than brute forcing [23, 24]. Although no hash function has been directly proven to be collision-resistant, some of them are *provably secure*, which means that finding collisions is proven to be at least as difficult as some hard mathematical problem (such as integer factorization or discrete logarithm) [20].

Property 1: Collision-resistance: A hash function H is said to be collision resistant if it is infeasible to find two values, x and y, such that $H(x) = H(y)$, but $x \neq y$.

One of many immediate applications for collision-free hash functions, is as a message digest. The hash of any string then serves as a fixed length digest, or unambiguous summary, for the message. For two given hash value outputs $H(x)$ and $H(y)$ from a collision-free hash function, it is safe to assume that their inputs must also be different. Therefore, knowing the hash value to a message serves as a guaranty that we can always check the integrity of the message by checking that it produces the same hash value. Furthermore, by just making minimal changes to the message, most hash functions will create a completely different hash, making it easy to spot a manipulation even for the human eye.

The next property to be discussed is the *hiding property*. In information theory *min-entropy* is a measure of how predictable an outcome is, and high min-entropy means that the distribution of the random variable has high variability. In other words, there is no particular value from the distribution that is more likely to be sampled. As an example, if r is chosen uniformly from among all the strings that are 256 bits long, then any particular string is chosen with probability $1/2^{256}$, which is an infinitesimally small value.

Property 2: Hiding: A hash function H is said to be hiding if a value r chosen from a probability distribution that has high min-entropy, makes it infeasible to find x when given $H(r|x)$.

This property has application in commitment schemes, where one can commit to a message solely by publishing the commitment of it that does not reveal the message itself. First, one picks a random value and concatenates it with the message to commit to. This will be used as an input to a hash function that is hiding and collision-free, and the hash is published as the commitment. Since the hash is collision-free, it is infeasible to find two different inputs that give that same hash. By publishing the hash one commits to that input. Because the message may be one of just a few possibilities, concatenating it with a random value guarantees with the hiding property the infeasibility of finding the random value that together with the message would yield the hash value and thereby disclose the message, which was committed to. At any time one can reveal the commitment by publishing the random value and message. Everyone can use this as input to calculate the hash that must correspond to the one that was published prior to the revelation.

The hash function, which is used most extensively in Bitcoin is *SHA-256*, which gives an output of 256 bits. It has all the properties described above and is used widely among a lot of applications, such as in TLS, SSL, PGP and SSH.

2.2 Hash pointers

A collision-free hash function can be used to make a data structure that points to where some information is stored together with a cryptographic hash of that information. This data structure is called a hash pointer. Whereas a regular pointer will only show from where some information can be retrieved, a hash pointer also gives a way to verify that the information has not been changed. Many data structures that use regular pointers such as linked lists or binary search trees can also be implemented with hash pointers.

It is of special importance for Bitcoin to take a look at the implementation of a linked list with hash pointers, referred to as a *blockchain*. In a regular linked list of blocks, each block contains some data and a pointer to the previous block in the list, creating a sequence of blocks. The only thing needed to change such a linked list of blocks into a blockchain, is to replace the pointers with a hash pointer. Each block does then not just contain information about where the value of the previous block is, but also a summary of that value through its hash, allowing for the verification that the value has not changed. As will be explained below, only the hash pointer at the head of the blockchain must be stored safely to guarantee the integrity of the whole blockchain.

This data structure allows for a simple implementation of a tamper-evident log. It allows data to be stored by appending new blocks of data to it. If somebody would alter the data in the log, it would be detected from the hash pointer at the head of the log in the following way.

If an adversary changes data of some block k, then by the property of collision-resistance, so has its hash, now denoted as k'. Thus the hash pointer of block $k+1$, which contains the hash of the entire previous block k, would not match up with the hash of the now altered block k'. The inconsistency between the altered block k' and block $k+1$ would therefore be detected.

As long as the hash pointer at the head of the list is in a place where the adversary cannot change it, the adversary cannot cover up his change by just changing all the following hashes of blocks.

2.3 Public-key cryptography

Elliptic curve cryptography is used to create public/private key pairs in Bitcoin. An elliptic curve is a set of points described in general by the equation $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$ to exclude singular curves. Furthermore, all elliptic curves are symmetric around the x-axis, as can be seen from the equation. Bitcoin uses the specific elliptic curve specified by the National Institute of Standards and Technologies (NIST) called *secp256k1* and is defined by the curve $y^2 = x^3 + 7$.

To perform binary operations of addition $(+)$ and multiplication $(\cdot)$ within the elliptic curve, a group over the points on the elliptic curve is defined. In order for G to be a group, addition must be defined in such a way, that the following four properties hold:

- *Closure*: if a and b are members of G, then $a + b$ is also a member of G,

- *Associativity*: $(a + b) + c = a + (b + c)$,
- There exist an *identity* element 0 such that $a + 0 = 0 + a = a$
- Every element has an *inverse*, that is for every a there exists a, b such that $a + b = 0$

Adding a fifth property:

- *Commutativity*: $a + b = b + a$

defines an abelian group.

As an example the set of integers $\mathbb{Z}$ with the commonly known addition and multiplication operators define an abelian group, while the set of natural numbers $\mathbb{N}$ is not a group, since there exists no inverse to the elements of $\mathbb{N}$. In a group, the identity element is unique, and furthermore for each element in the group, the element is guaranteed to have a unique inverse element [16]. Over the elliptic curve a group will be defined as follow:

- Elements of the group are the points on the elliptic curve,
- The identity is a point defined as the point at infinity 0,
- The inverse of a point P is the one symmetric about the x-axis,
- Addition is defined in the following way. Given three aligned, non-zero points P, Q and R, their sum is $P + Q + R = 0$. Alignment means that a straight line can be drawn through all the aligned points.

Since addition only requires the points to be aligned, without considering any ordering of the elements, the definition of addition gives rise to both associativity and commutativity and hence defines an abelian group.

From the rule of addition and the fact that it defines an abelian group, $P + Q + R = 0$ can be rewritten as $P + Q = -R$. This implies that for two aligned points P and Q for which the straight line through these points also passes another point R on the curve, the sum of P and Q is defined as the inverse element of R, namely the reflection of R on the x-axis.

The definition of addition works except for special cases involving the point at infinity and intersection multiplicity.
If $P = 0$ or $Q = 0$, then no straight line can be drawn through the point 0, since it is not part of the xy-plane. But since it is defined as the identity element, $P + 0 = P$ and $Q + 0 = Q$.

If $P = -Q$, then $P + Q = P + (-P) = 0$, from the definition of inverse.
If $P = Q$, then there are infinitely many straight lines passing through the point. Considering a point $Q' \neq P$, and letting Q' approach P, then as Q' tends toward P the line passing through those two points becomes the tangent of the curve at the point P. Hence $P + P = -R$, where R is the intersection between the curve and the line tangent to the curve at P.
Finally, if $P \neq Q$ but the line only intersects the curve in those two points, then the line must in fact be tangent to the curve at either P or Q. If P is the tangent point, then from above $P + P = -Q$. Rearranging therefore gives $P + Q = -P$. If instead Q was the tangent point, then $P + Q = -Q$.

Scalar multiplication can naturally be defined as $n * P = P + P + P + + P$ (n times). Using these definitions, addition can be performed on any point on the curve. The rules describing addition geometrically on elliptic curves can be transformed into a set of equations. Those details and equations will not be given since it is not relevant for the understanding of Bitcoin and hence out of scope for this thesis.

By using the algebraic equations for addition, given n and P, then $Q = n * P$ can be easily computed, which for historical reasons is referred to as the exponential problem in public/private key cryptography. If instead P and Q were given, then finding n is called the logarithm problem. The exponential and logarithmic problem refers to the RSA encryption scheme, where instead of multiplication the encryption uses exponential operators.

If the domain of the elliptic curve is reduced to span over a finite field of prime order, then scalar multiplication remains "easy", while no efficient general method for computing discrete logarithms on conventional computers is known. Public-key cryptography base their security on the assumption that the discrete logarithm problem over carefully chosen groups has no efficient solution. This duality is the key to elliptic curve cryptography.

Defining an elliptic curve produced over a finite field is as follows:

$$y^2 = x^3 + ax + b, \quad 4a^3 + 27b^2 \neq 0, \quad \text{over } (\mathbb{F}_p)^2$$

Specifically for the secpt256k1 used in Bitcoin:

$$y^2 \bmod p = (x^3 + 7) \bmod p$$

where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, so a very large prime number.

The elliptic curve cryptography then works as follows. By starting with a random number k which acts as the private key, it is multiplied by a predetermined point on the curve G, called the *generator point*. The generator point is part of the specification in secp256k1 and the same for all keys in Bitcoin. This produces another point on the curve which will correspond to the public key K, so $K = k * G$. Since reversing this equation is a "hard" problem, it is safe to share the public key K with anyone, as it cannot be used to reveal or calculate the private key k.

Bitcoin does actually not use elliptic curve cryptography to encrypt data. Instead it uses the encryption's signing mechanism ECDSA to create signed transactions that proves the knowledge of the private key k corresponding to the public key K used to derive the Bitcoin address.

2.4 Digital signatures

Without giving a lot of details, the signing mechanism uses a hash-value h of a message m to be signed. The hash h must be truncated so that the bit length of h is the same as the bit length of the order of the subgroup n. The subgroup contains all the points on the elliptic curve generated by G, meaning all the numbers that can be generated as multiples of G. This is part of the specification by choosing an elliptic curve and base point. It generates a secret number p in $\{1, 2, \ldots, n-1\}$, which is a number similar to a private key. This secret is used to calculate the point $P = p * G$, which uses the similar calculation as for public keys. Then the number $r = x_P \bmod n$ is calculated, where x_P is the x-coordinate of the point P.

Another number s is generated by using the signer's private key k, the number r from above and the hash-value h in its calculation, such that $s = p^{-1}(h + r * k)$. The pair (r, s) is then the signature of the message m.

To verify a signature, the recipient will need the signer's public key K, the truncated hash-value h obtained by hashing the received message m and truncating as above, and the signature (r, s). The verification algorithm will use the fact that from these values provided, the point P can be calculated and r must equal the x-coordinate of that point.

The integers $u_1 = s^{-1}h \bmod n$ and $u_2 = s^{-1}r \bmod n$ are calculated. Then $P = u_1G + u_2K$ and the x-coordinate can be compared to r. If they are not equal, then either r or s is wrong (or both). This would imply that no correct signature was provided, the public key K is wrong, or the hash h obtained from the message is wrong, meaning the message was alternated after signing it.

The reason that P equals $u_1G + u_2K$ will become clear from the definition of the public key, namely that K equals kG.

$$\begin{aligned} P &= u_1G + u_2K \\ & u_1G + u_2kG \\ & (u_1 + u_2k)G \end{aligned}$$

Using the definition of u_1 and u_2, P can be rewritten to

$$\begin{aligned} P &= (s^{-1}h + s^{-1}rk)G \\ P &= s^{-1}(h + rk)G, \end{aligned}$$

where $\mathrm{mod} n$ could be omitted since the subgroup generated by G is of order n.

The last step follows from the definition $s = p^{-1}(h + rk)$. Multiplying by p on both sides and dividing by s gives $p = s^{-1}(h + rk)$, which substituted into the last equation yields $P = s^{-1}(h + rk)G = pG$. This also shows why first of all it is important to keep the secret p really secret and to use different secret values for p for each signature! If two signatures for two hashes where using the same p value, then the secret p could be extracted from the equations above.

The details on how this could be exploited will be skipped. This signing method is used extensively in Bitcoin. After signing a transaction, it cannot be altered anymore, meaning that amount or destinations cannot be changed, at the same time proving that the person spending the bitcoins knows the private key of that Bitcoin address containing the funds.

Part I

The system of Bitcoin

I've been working on a new electronic cash system that's fully peer-to-peer, with no trusted third party.

Satoshi Nakamoto

3 A brief history of cryptocurrencies

Money is one of the oldest technologies humans have developed, and Bitcoin is an innovation in the technology of money. In order to have a discussion about what Bitcoin is, and why it works, one first need to have a conversation about what money is, and why it works.

While exchanging value from a person A to a person B in a trust-less manner can be done by meeting physically and then handing over the good of value, e.g. in form of currency or precious metals such as gold, payment systems, such as for example the banking system, must be used with complete trust in the system. The bank, government or corporation of a payment system is in control of the funds and value in transactions, and therefore must be trusted to handle a transaction correctly for the client, most often protected by the law. But this does involve risk, for example the bankruptcy of a bank, government or corporation involved, or the installation of capital controls which would limit or prohibit withdrawing funds by law.

Historically, a good currency typically consists of the following properties:

- scarcity,

- divisibility and
- fungibility.

Divisibility and fungibility means that it is easy to divide the currency into smaller or greater units and that any unit of the same currency is indistinguishable from another.

Scarcity means that the currency is limited in supply, immediately implying that it can also not be easily counterfeited. Government issued currencies such as the Euro and US Dollar contain watermarks and other physical security protections to guarantee that only the central bank is able to print and issue the currency. But because governments can print new money depending on laws that may change over time, all government issued fiat money, such as the Euro and the US Dollar, derive their scarcity property solely from government regulation and law.

As enough people trust those governments to only issue new money with good policies in place protecting the money supply, it is accepted as a currency even without an implicit scarcity property. Historically though, most fiat currencies have experienced hyperinflation making the fiat money worthless in a very short time [5].

Early digital money was either issued directly through a bank, such as in DigiCash [7], which obtained its value through a direct peg to the US Dollar, or issued through a company such as e-Gold, that would store gold in its vault and issue digital cash up to the value of that gold.

Therefore the value of money does not come from the precious metals, and it does not come from the government issuing currencies. It comes directly from the economic activity it creates through exchange. It comes from the social bonds it creates. The value is derived through the believe of the value it has. Money is a great societal illusion.

A radically different idea is to allow a digital money to be its own currency, issued and valued independently of any other currency or commodity. For this to work, you would likely need to create a digital currency that is scarce by design. One way is a system that requires solving computational problems, or "puzzles", to mint new money. This is what happens in Bitcoin.

Bitcoin is the first so called *decentralized cryptocurrency* and was invented in 2008 with the publication of a paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" [18], written under the alias of Satoshi Nakamoto. A cryptocurrency combines techniques from cryptography and distributed systems to create a payment system, which is decentralized and peer-to-peer.

This basic idea of creating digital objects that have some value through computational puzzles was already proposed in 1992 by Dwork and Noar [10], but as a possible solution to email spam. For the recipient to accept your email, it would require you to find a solution to a computational puzzle that would take a few seconds to solve. A similar idea was also discovered by Adam Back in 1997, in a proposal called Hashcash [4]. These computational puzzles need to have some specific properties:

- be specific to the email,
- the receiver must easily verify the solution,
- each puzzle must be independent of the others, and
- the difficulty of the puzzles must be adjustable.

These properties can be achieved by using cryptographic hash functions discussed in Chapter 2. Bitcoin uses principally the same puzzle as Hashcash.

Why did these proposals to combat email spam never take off? Maybe the development of spam filters was a good enough solution, and the remaining spam not a big enough problem for spending computing cycles on combating them. In the end, mybe these proposals would also not have worked. Most spammers today use so called botnets, which are large groups of other people's computers, to send spam.

In 1991 Haber and Stornetta published a paper describing a method for secure timestamping of digital documents [15]. The idea behind timestamping is to give an indication on when a document first came into existence. But more importantly, it accurately dictates the order of relative creation. If one document came into existence before the other, then the timestamp will reflect that. In Haber's and Storenetta's scheme, there is a timestamp service. It signs received documents together with the current time and a pointer to the previous document, and issues a certificate containing this information.

Instead of linking to a location, the pointer links to a piece of data. If the data changes, then the pointer becomes implicitly invalid, guaranteeing the integrity of the timestamping. Each certificate thereby essentially encumbers the entire history of documents up to that point.

A later proposal gave an efficiency improvement. It is possible to bundle documents together into blocks and link blocks together as a chain. Within each block, documents would be linked together in a tree structure, rather than linearly. This data structure became a key component of Bitcoin, acting as its backbone. By using a Hashcash puzzle to delay how fast new blocks can be added to the chain, a collection of untrusted nodes can keep track of blocks in Bitcoin, rather than having to rely on trusted servers to do it.

Bitcoin uses the idea of regulating the creation of new currency units by computational puzzles, combined with securely timestamping a ledger of transactions inside blocks to prevent double spendings.

There were early proposals of digital money combining these two ideas though. One is called b-money by Wei Dai in 1998 [9]. In b-money, anyone can create money using a hashcash similar system. There is a peer-to-peer network just as in Bitcoin. Each node maintains a ledger, but it is not globally defined. Instead, each node has its own version of what it thinks everyone's balance is. B-money does also rely on timestamping services to sign off on the creation and transfer of money. A similar proposal was called Bitgold by Nick Szabo [21] in 2005. Neither idea ever took off. They ignored issues that may or may not have been solvable. The first is how to resolve disagreements about the ledger. Another is how hard the computational puzzles should be to mint a unit of currency, since hardware is becoming drastically faster with time.

The creator of Bitcoin, Satoshi Nakamoto, withdrew from the public in April of 2011, leaving the responsibility of developing the code base and network to a thriving group of volunteers. The identity of the person or people behind Bitcoin is to date unknown. However, neither Satoshi Nakamoto nor anyone else has individual control over the Bitcoin protocol, which operates solely based on fully transparent mathematical principles, open source code and the consensus established among participants. The name "Bitcoin" describes both the payment system, spelled with a capital B, as well as the unit of account within this payment system, spelled with a small b as bitcoin.

Bitcoin is the beginning of something great: a currency without a government, something necessary and imperative.

Nassim Taleb

4 Transaction system

With all the prerequisites in place, lets start by looking at how transactions can be broad casted in a messaging system.

We begin by building a simplified version of a cryptocurrency, implementing a data structure for transactions. We will develop it further in different steps throughout this and the next chapter and improve its security properties. Finally, evolving the cryptocurrency into a model of a permission-less and decentralized cryptocurrency, similar to how Bitcoin works. Differences between our simplified cryptocurrency and Bitcoin will always be mentioned when applicable.

4.1 A simple cryptocurrency: Version 1

The first version of the cryptocurrency will be denoted as AliceCoin. It consists of just two rules. The first rule says that a designated entity called Alice can create new coins at will, which automatically belong to herself. When Alice decides to create new coins, she just has to generate a unique serial number for the new coin, denoted `[SerialNumber]`. With her secret signing key, she then has to digitally sign the string `GenCoin [SerialNumber]`. This string, together with her signature, amounts to a new coin, which is uniquely identified through its serial number. Anyone can verify the validity of the coin by checking Alice's signature against the statement generating that coin.

The second rule describes how anybody owning a coin can transfer it to somebody else by provably changing the ownership of said coin to the new one. This is done by using cryptographic operations in such a way that transactions spending coins must be signed by the owner of those coins. Alice can pay her newly created coin by signing a transfer statement `Send [H( )] to Bob`, where `[H( )]` contains a hash-pointer to the `GenCoin` statement of her coin. The hash-pointer solely consists of taking the hash value of that whole `GenCoin` statement. Bob will see that the transaction sending him the coin was signed by Alice. From rule one Bob can see that indeed Alice was the owner of the coin prior to him, since the transaction references her own signed `GenCoin` statement.

Anybody who owns a coin and wants to spend it, must sign a similar transfer statement. However, instead of referencing a `GenCoin` statement, they will just reference a transfer statement in which they received the coin. For example, assume Bob wants to transfer to Chuck the coin he received from Alice. He creates the transfer statement `Send [H( )] to Chuck`, where this time `[H( )]` is the hash pointer to the transfer statement Bob received from Alice, or any other valid transfer statement he received. Finally, Bob signs this statement. Now Chuck can prove to anybody that he owns the coin. They can follow the chain of hash pointers back to the coin's creation and in each step verify that the rightful owner signed the transfer.

But Bob discovered a fundamental flaw in AliceCoin. Nothing can stop him from double-spending his coins. If Bob, without telling anybody else, passes his coin on to Chuck with a signed transfer-statement, he could create another signed transfer-statement that sends the same coin also to Charlie. To Charlie, that transaction would appear perfectly valid, making him the owner of this coin. Thus Bob would have successfully paid the same coin to both Chuck and Charlie.

To circumvent this problem of double-spending coins, cryptocurrencies must rely on ledgers. Before presenting an improved version of AliceCoin using a ledger, a more in-depth description of the data structure of Bitcoin transactions will be presented. The data structure is very similar to the mechanism used in AliceCoin for transferring coins.

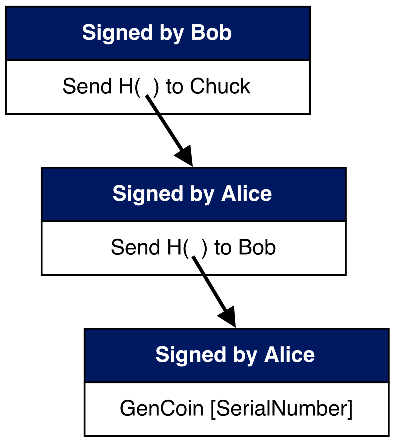


Figure 4.1.1: A simple chain of transactions in AliceCoin. The bottom is the `GenCoin` transaction, which issues the coin. The middle transaction sends the coin to Bob by pointing to the `GenCode` transaction with a hash pointer and changing the ownership from Alice to Bob by providing Alice's signature. The last transaction sends the coin to Chuck, who can verify that every transaction in the chain rightfully changed the ownership of the coin by checking the signatures provided in each transaction by the owner in that time.

4.2 Data structure for Bitcoin transactions

Transactions in Bitcoin are messages defined as a data structure. They are chained together, such that a new transaction will spend bitcoins from unspent transactions that the spender has received. Every transaction has an ID, which simply corresponds to its hash value. A transaction consists of a source of funds, called *inputs*, with *unlocking scripts* for each of those inputs. A transaction also has *outputs* that define the destinations and their amounts of funds to receive. Each output also contains a *lock script*. The lock scripts contain instructions that determine how the transferred bitcoins of an output can be spend again. To uniquely identify inputs and outputs, they are listed within the transaction with unique index numbers, starting at zero for each input and output.

Typically, bitcoins are transferred to Bitcoin addresses. A Bitcoin address is a unique number similar to a bank account number, or like an email address. In the output of a transaction that sends funds to a Bitcoin address, the lock script must contain instructions requiring a proof of ownership of said Bitcoin address. In other words, the lock script implicitly incorporates the recipient's Bitcoin address by requiring a proof of ownership of exactly that address whenever the recipient wants to spend those bitcoins with a new transaction.
Generally speaking, a lock script defines requirements on who or how bitcoins can be spend

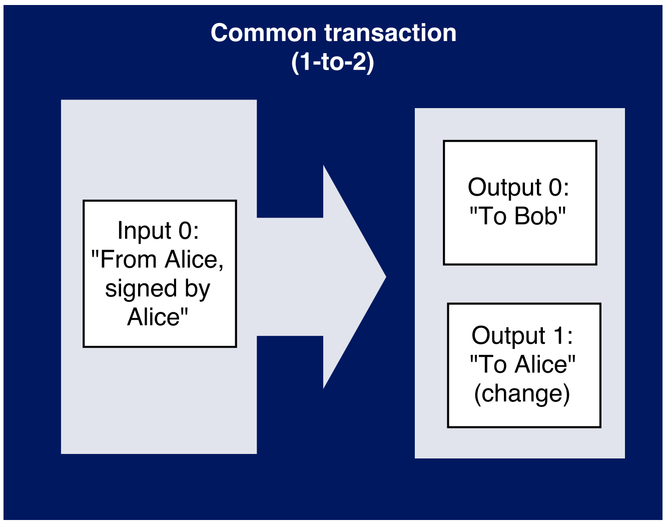

Common transaction
(1-to-2)
Input 0: "From Alice, signed by Alice"
Output 0: "To Bob"
Output 1: "To Alice" (change)

Figure 4.2.1: One example of the use of inputs and outputs in a common Bitcoin transaction. Alice creates a transaction paying some bitcoins to Bob. In this case the amount of bitcoins Alice uses in Input 0 is greater than what Alice wants to pay to Bob in Output 0. Therefore she includes another output Output 1 which pays back the left-overs to herself again.

again, and the unlocking script of the transaction input that spends those bitcoins must satisfy these requirements. Both the lock and unlocking scripts are programmed using a programming language called Script, see Chapter 6. Although unlocking scripts can potentially also be written as script programs, they usually only contain the information needed to satisfy the conditions determined by a lock script.

The inputs of a new transaction determine which funds to spend and must contain the transaction IDs and output indices to unspent transaction outputs. The whole amount of bitcoins used by inputs will be spend. The sum of all those funds used by the inputs must at least equal the total amount the spender wants to transfer. Therefore, any “left-overs” from using too many bitcoins with the inputs must in that same transaction be sent back to the sender’s own address. This is done by including an output that contains a lock script that locks those left-overs to the sender’s own Bitcoin address. This is similar to how a pocket wallet works. If a client has to pay 1.20 Euros for a coffee, but only has a one Euro and a 50 cent coin in his wallet, then he will use both coins for a total of 1.50 Euros, paying 1.20 Euros and expecting to receive 30 cents as change for the left-over.

Bitcoin transactions can use any number of inputs and outputs. Figure 4.2.2 shows transac-

Table 4.2.1: The data structure of a Bitcoin transaction

Size	Field	Description
4 bytes	Version	Specifies which rules this transaction follows
1-9 bytes	Input counter[a]	Specifies how many inputs are present
Variable[b]	Inputs	One or more transaction inputs
1-9 bytes	Output counter[a]	Specifies how many outputs are present
Variable[b]	Outputs	One or more transaction outputs
4 bytes	Locktime	References a Unix timestamp or block number

[a] The value is an integer.
[b] The size depends on the number of inputs or outputs.

tion types that aggregate inputs together with only one potentially bigger output, or split one input in many smaller outputs.

To include a transaction fee, whose purpose is explained later, a transaction just has to use a greater amount of bitcoins from the inputs than it specifies in the outputs. The total net difference will then make up the fee for that transaction. Figure 4.2.3 shows a chain of Bitcoin transactions, ignoring the lock and unlocking scripts.

The data structure of Bitcoin transactions can be seen from Table 4.2.1. The locktime can define the earliest time that a transaction becomes valid. But when set to zero, no locktime is present and the transaction is valid immediately.

Transaction inputs are pointers to unspent transaction outputs. They point to them by referencing the transaction hash and index from that output. To spend an unspent output, the input also includes an unlocking script that satisfies the spending conditions set by the unspent output. The structure of a transaction input is shown in Table 4.2.2.

The sequence number is typically just set to the value 0xFFFFFFFF. It was initially thought to be used for replacing transactions that were still unconfirmed. Instead, there is now a proposal that will change its purpose, making it much more meaningful. The sequence number will be repurposed to prevent validation of a transaction until a certain age of the spent output, also referred to as the "maturity" of the transaction. This means that a transaction remains invalid until for each of its input, the time defined with each sequence number has passed for its corresponding output that it spends. So, if e.g. the sequence number for some input is set to 7 days, then the transaction can first become valid after the transaction that it spends is at least 7 days old. All the details regarding how the notion of time can be interpreted in this decentralized

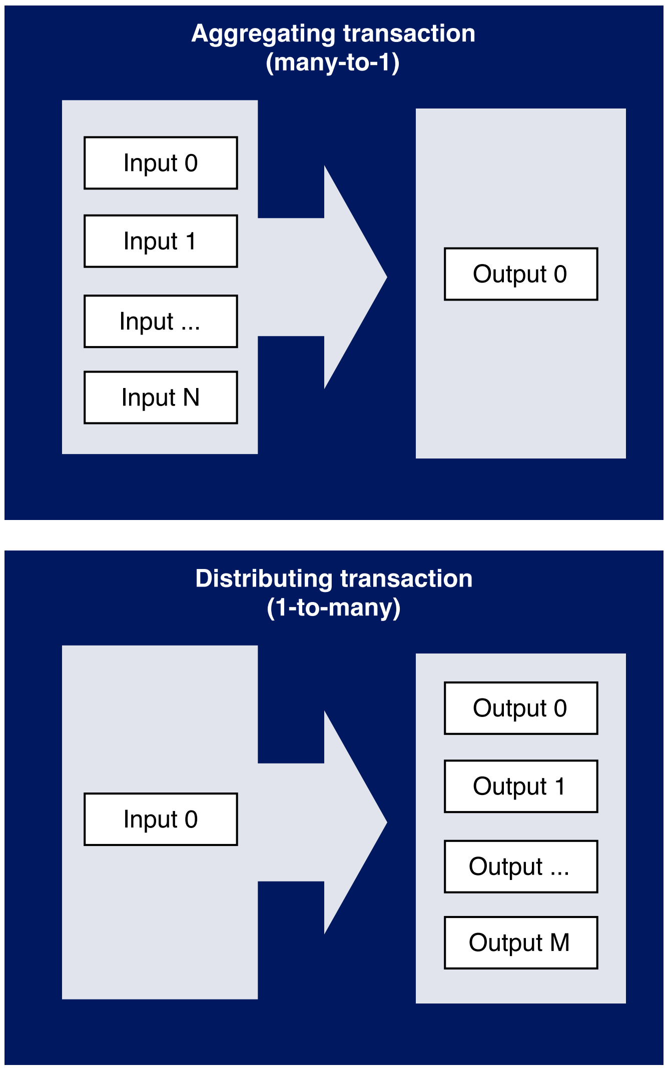


Figure 4.2.2: Aggregating and distributing transactions. The first transaction type bundles together many inputs with only one potentially greater output. The other transaction type splits up a single input into many smaller outputs.

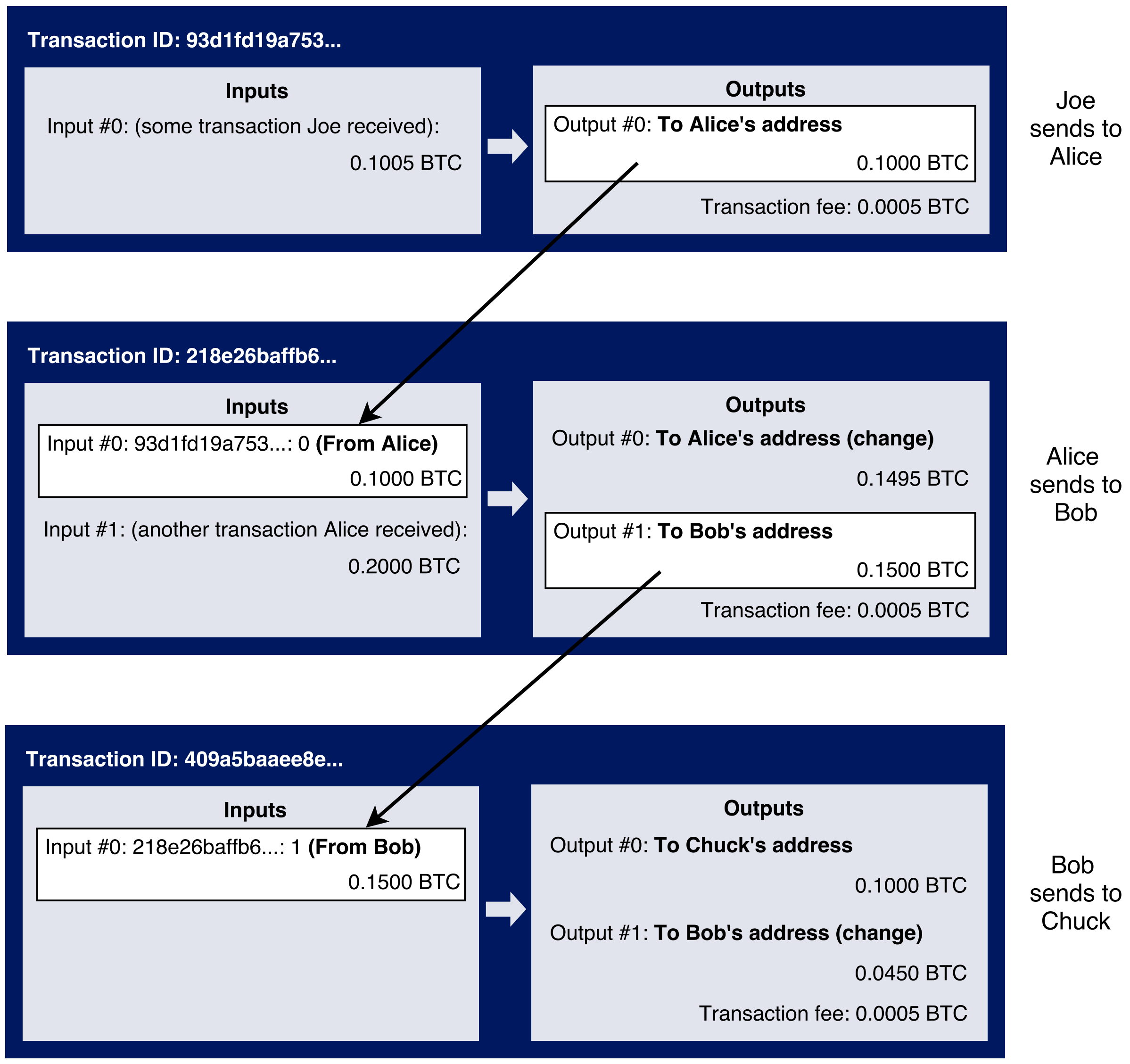

Transaction ID: 93d1fd19a753...
Inputs
Input #0: (some transaction Joe received):
0.1005 BTC
Outputs
Output #0: To Alice's address
0.1000 BTC
Transaction fee: 0.0005 BTC
Joe sends to Alice
Transaction ID: 218e26baffb6...
Inputs
Input #0: 93d1fd19a753...: 0 (From Alice)
0.1000 BTC
Input #1: (another transaction Alice received):
0.2000 BTC
Outputs
Output #0: To Alice's address (change)
0.1495 BTC
Output #1: To Bob's address
0.1500 BTC
Transaction fee: 0.0005 BTC
Alice sends to Bob
Transaction ID: 409a5baaee8e...
Inputs
Input #0: 218e26baffb6...: 1 (From Bob)
0.1500 BTC
Outputs
Output #0: To Chuck's address
0.1000 BTC
Output #1: To Bob's address (change)
0.0450 BTC
Transaction fee: 0.0005 BTC
Bob sends to Chuck

Figure 4.2.3: A chain of Bitcoin transactions. Output 0 from the first transaction is spent by specifying its transaction ID and index in the input for the second transaction of the chain. Each transaction includes a fee, which is the difference of the total amount of bitcoins from the inputs and outputs.

Table 4.2.2: The data structure of an input to a Bitcoin transaction

Size	Field	Description
32 bytes	Transaction hash	Hash of the transaction containing the unspent output to be spent.
4 bytes	Output index	The index number of the unspent output to be spent
1-9 bytes	Unlocking script size	Size of the unlocking script in bytes
Variable	Unlocking script	A script that satisfies the conditions set by the lock Script of the unspent output
5 bytes	Sequence number	Typically set to 0xFFFFFFFF

Table 4.2.3: The data structure of an output to a Bitcoin transaction

Size	Field	Description
8 bytes	Amount	Bitcoin value in Satoshis[a].
1-9 bytes	Lock script size	Size of the lock script in bytes
Variable	Lock script	A script defining the conditions needed to spend output

[a] The bitcoin unit was chosen to represent a value of 10^8. A Satoshi is the smallest possible denomination and therefore 0.00000001 bitcoin.

system will be explained in later chapters. The sequence number is also referred to as a relative locktime, since it depends on the relative time that has passed since its parent was confirmed. It is therefore in contrast to the locktime, which is an absolute locktime for some specific date to be reached. We will return to the sequence number and locktime in Chapter 6, in which this new interpretation of the sequence number will be assumed to be part of the consensus rules of the Bitcoin network.

A new transaction will create unspent transaction outputs that each can be referenced as an input to new transactions spending those bitcoins again. As already mentioned, the output consists of two parts: the amount to send, and a lock script. Sending someone bitcoins therefore consists of creating an unspent transaction output registered to their bitcoin address, available for them to spend.

The structure of a transaction output is shown in Table 4.2.3.

Every node in the Bitcoin network that receives a new transaction will validate it by running both the lock and unlocking script simultaneously and check that the unlocking script satisfies the condition from the lock script.

As of today, most transactions have the form “Alice pays Bob”. But transactions are not limited to the form of “Alice pays Bob” and can be programmed to contain a lot of different conditions, many of which will be explored later. In the following we will investigate how account numbers can be generated and how exactly the proof of ownership of a Bitcoin address is provided.

4.3 Bitcoin Addresses: The Account numbers of Bitcoin

Figuratively, Bitcoins are digitally attached to an account by the lock scripts of transactions as described above. The balance of a Bitcoin account is determined by checking all ingoing and outgoing transactions with lock scripts to that account. Bitcoins are therefore just a unit of account, and the owner of an account is in control of creating transactions, which can transfer bitcoins from his account to another. The accounts in Bitcoin are refered to as *public Bitcoin addresses*, and in a transferred sense they are similar to a bank account number or email address. But Bitcoin addresses can be created offline without a connection to the internet and do not need to be registered. A public Bitcoin address is the hash-value of a public key, for which there exists a corresponding private key. Bitcoin uses elliptic curve cryptography to create such a private key. The hash-value consists of first using the SHA256 hash-function on the public key and afterwards using the hash-function RIPEMD160 on that hash to create a 160-bit double-hashed value.

To derive a public Bitcoin address from this hash, a version prefix of 1 byte is added to differentiate e.g. between the public Bitcoin address with prefix zero (0x00) or a similarly double-hash encoded private key with prefix 128 (0x80), referred to as the private Bitcoin key. But many more prefixes exist. Next, a checksum is calculated from the version prefix concatenated with the double-hash value of the public key by using the SHA256 hash function twice on it. From this hash value only the first 4 bytes serve as the checksum and are appended to the double-hash value of the public key with version prefix. This corresponds to a complete Bitcoin address. However, for readability purposes a Bitcoin address is generally presented in *Base58Check encoding*. Base58Check encoding uses all alphanumeric characters, but without the characters I (eye), l (one), O (oh) and o (zero), to reduce the possibility of misspelling by characters having a similar appearance.

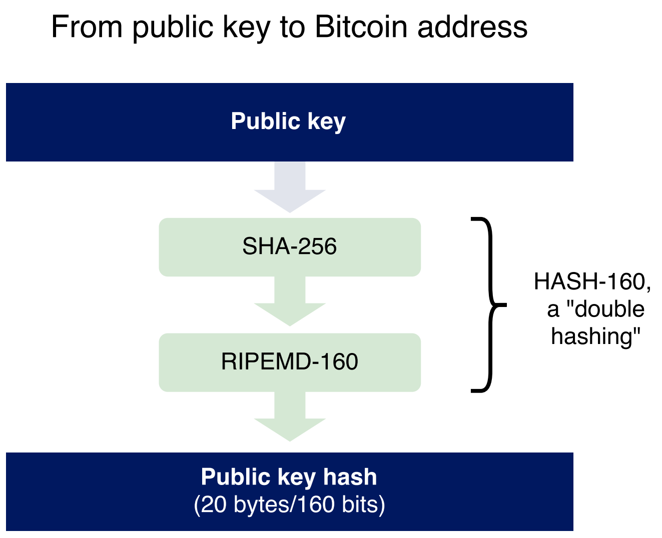

From public key to Bitcoin address
Public key
SHA-256
RIPEMD-160
HASH-160, a "double hashing"
Public key hash (20 bytes/160 bits)

Figure 4.3.1: Public Key to Hash conversion. The public key is first hashed using SHA256 and afterwards RIPEMD160, returning a double-hash value of length 160 bits.

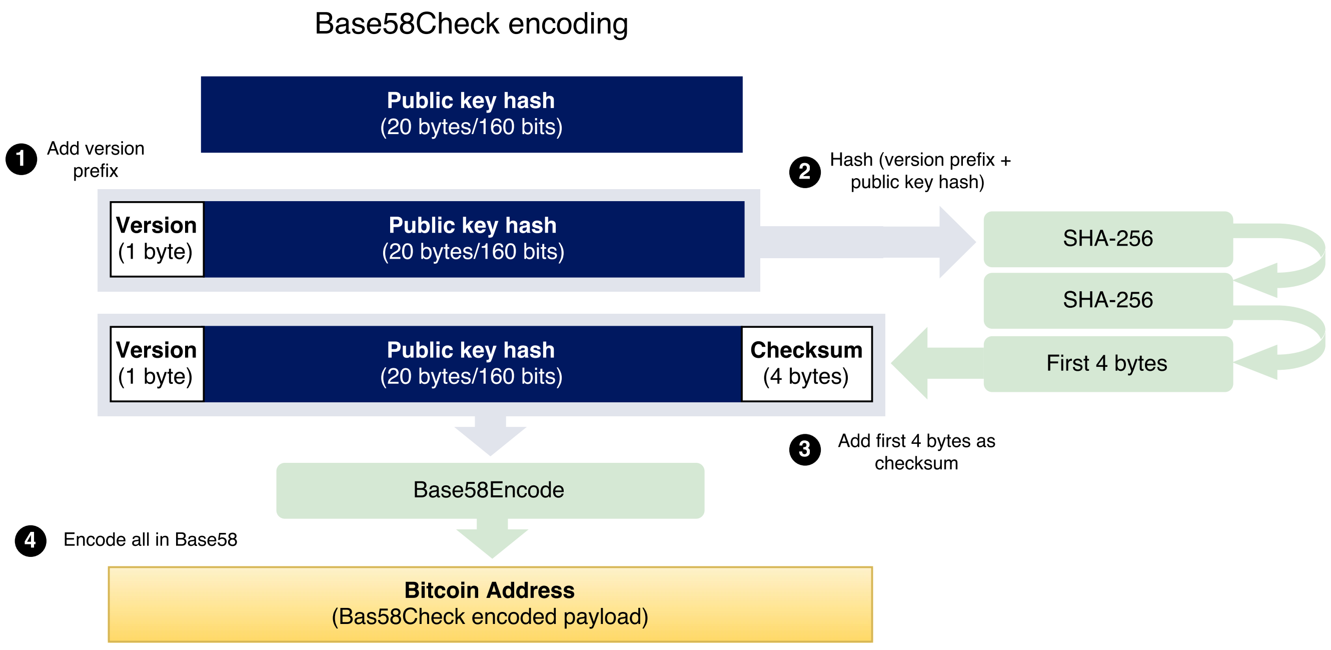

Base58Check encoding
Public key hash (20 bytes/160 bits)
1 Add version prefix
2 Hash (version prefix + public key hash)
Version (1 byte)
Public key hash (20 bytes/160 bits)
SHA-256
SHA-256
First 4 bytes
Version (1 byte)
Public key hash (20 bytes/160 bits)
Checksum (4 bytes)
3 Add first 4 bytes as checksum
Base58Encode
4 Encode all in Base58
Bitcoin Address (Bas58Check encoded payload)

Figure 4.3.2: Schematic Base58Check encoding procedure. A version prefix and checksum is added to the public key hash and represented in Base58 encoding.

Because of the version prefix, every standard Bitcoin address that is represented in Base58Check will start with the character 1, while e.g. a private Bitcoin key in Base58Check encoding will start with the character 3. Due to the checksum appended at the end of the Bitcoin address, a Bitcoin client program can make sure that there are no misspellings in the address of a transaction before propagating it to the network.

4.4 Proof of ownership of a Bitcoin address

Using a digital signature algorithm from elliptic curve cryptography, an unlocking script can provide the proof of ownership of a Bitcoin address that is required by a lock script. First of all, in a standard Bitcoin transaction, the lock script will require that its unlocking script provides the public key to the Bitcoin address that received the funds. Furthermore, it must provide that key's signature produced for the spending transaction. This makes a Bitcoin transaction comparable to a check, since anyone can create the template for a transaction, but only the owner of the funds being spend can make the transaction valid by signing it.
When a transaction is signed, it cannot be changed without invalidating it. Changing anything will make the hash of the transaction different from the hash used for the signature. But there exist different types of signatures, which allow specific parts of a signed transaction to be changed without invalidating the signature. Also, since the lock and unlocking scripts are programmable, much more complex transactions in form of contracts are possible, which will be described in Chapter 6.

Bitcoin is a remarkable cryptographic achievement and the ability to create something that is not duplicable in the digital world has enormous value.

Eric Schmidt

5 Decentralization using a distributed ledger

This chapter introduces many of the mechanisms of how and why Bitcoin works in a decentralized and peer-to-peer way. The parts that will be of focus in this chapter is to solve the double spending problem discussed in Chapter 4. In regard to the aim of this thesis however, the description of these mechanisms are not necessarily required to know about, as it does not directly involve the transaction structure or programming of the outputs, which will be presented in Chapter 6.

5.1 Use of a public ledger with the simple cryptocurrency: Version 2

After having discussed the details on how transactions work in Bitcoin and having understood its data structure, we can return to take a look at how Bob can improve Alicecon. To solve the double-spending problem, Bob will propose a new cryptocurrency that we will call Bobcoin. It is built off of Alicecoin, but will involve more complication in terms of its data structure. First of all, Bob will be a designated entity that publishes a history of all transactions that have

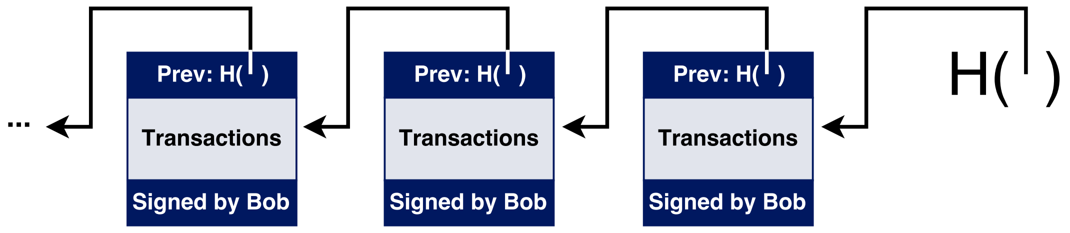


Figure 5.1.1: The blockchain of Bobcoin. Bob adds transactions to a new block, which have a hash pointer to their last block in the list. By always securely storing the hash of the latest block (the head of the list), it makes the blockchain tamper-evident. Each block is furthermore signed by Bob, acting as the sole authority deciding which transactions are included.

happened. For this he will use a blockchain, which he will digitally sign using his public-private key pair. The blockchain will contain a series of data blocks, each of which can contain one or more transactions. When a transaction is included in a block, that block will contain the ID of the transaction and its data. Furthermore, every block contains a hash pointer to the previous block. Bob will digitally sign that final hash pointer, which represents this entire structure, and will publish the signature along with the blockchain. In Bobcoin, a transaction only counts as confirmed if it is in the blockchain. Anybody can verify that a transaction was accepted by Bob by checking his signature on the block that the transaction appears in. Bob makes sure that he does not accept a transaction that attempts to double-spend an already spent coin, which could be spotted easily by anybody looking inside the blockchain. Furthermore, the blockchain structure prevents Bob from being able to change his mind about the history of transactions. If he wants to add to, change or remove a transaction in the history, he would have to change the block within the blockchain that contains the transaction. But this would affect all of the following blocks due to the hash pointers. As long as someone is monitoring the latest hash pointer that Bob publishes, the change would be obvious and easy to spot. So a blockchain makes it easy for any two parties to verify that they have observed the exact same history of a transaction. It is important to note that, until Bob has added a transaction into a new published block, participants are not safe in accepting the transaction, since it could otherwise still be double spend. A transaction that is otherwise valid, but not in the blockchain, may never be added to the blockchain if Bob instead adds a transaction that spends the same coins.

So far Bobcoin works, in the sense that participants can see which coins are valid. It prevents double-spending, because everyone can look into the blockchain and see all of the valid transactions and spot potential changes made to blocks Bob already signed and published. But the problem is Bob himself. Even though he is unable to create fake transactions, since he cannot forge other people's signatures, he just still has too much influence otherwise. He can prevent transactions originating from targeted users to ever be added to a block, thus denying them service and making their coins useless. He can also create as many new coins as he wants, or suddenly bring the system to a halt if he gets bored of maintaining it. The problem is centralization. As described in the introduction, many early cryptocurrencies that relied on a central authority largely failed to take off. Although this happened for many reasons, in hindsight it appears that it is difficult for cryptocurrencies to achieve acceptance when they rely on a centralized authority. Arguably, one of the main reasons to Bitcoins success, is that it does not rely on any centralized authority to keep track of the balances in each account or to authorize valid transactions.

But to achieve this level of decentralization, we need to figure out a way for all users to agree upon a single published blockchain acting as the history to all transactions that have occured in the system. Decentralization is an important concept, which is not unique to Bitcoin. But trade offs are possible, and almost no system is purely decentral or purely central. Although email is fundamentally a decentralized technology, where anybody can operate an email server of their own, in reality a large portion of its users rely on a small number of centralized webmail providers that dominate the space. Similarly, while Bitcoin as a technology is decentralized and even encourages it, services like Bitcoin exchanges that let users convert bitcoin into other currencies may be centralized.

Returning to our hypothetical cryptocurrency, what if Chuck sees the problem of the centralization of Bobcoin and wants to decentralize it? Then he would have to look into distributed consensus protocols, which establish agreement among a number of nodes or processes for a single value.

5.2 General consensus protocols

In consensus protocols, nodes are participants in the network that must agree upon some value to establish consensus. Those networks are not centrally dictated (otherwise establishing consensus would be trivial), and there is the notion of *honest*, *faulty* and *malicious* nodes. Honest nodes follow rules, which the protocol dictates. Faulty nodes could be ones that give no answers because they have halted. Malicious nodes can even operate with their own rules disfavouring the consensus protocol.

Given n nodes that each have an input value, and some of the nodes may be faulty or malicious, the distributed consensus protocol has the following properties:

- It must terminate with all honest nodes agreeing on the same value.
- The value must have been proposed by an honest node.

In a cryptocurrency, the distributed consensus protocol must bring the nodes to agreement on exactly which transactions that were broadcasted become part of a single, global ledger containing no double spends. For that purpose, transactions can be bundled inside blocks locally by each node and eventually become their proposal value to the distributed consensus protocol. In that way, consensus is established in a block-by-block basis and at any given time all nodes in the peer-to-peer network have a ledger consisting of a sequence of blocks, each containing a list of transactions that they build consensus around. But in a peer-to-peer network there is no notion of a global time, which heavily constrains the set of known algorithms that can be used in the consensus protocol. The lack of global time leads to an impossibility result originating from the *Byzantine Generals Problem*.

Theoretical results

In the famous Byzantine Generals Problem, a group of generals of the Byzantine army are physically separate from each other but must agree upon a specific day and time on which jointly to attack a city. Only a united attack will be successful. The problem arises since there may exist persons acting as Byzantine generals, while in fact belonging to the enemy. Those faulty generals can send a message to one part of the generals with a time and day to attack, while sending a different time or day to the other part of generals. Analogous to the case of digital currencies, such a faulty general corresponds to someone who sends coins to some person and then that same coin, which is already spent, to another person. From this problem follows impossibility results showing that, under specific conditions, and with only a third or more of the nodes being malicious, achieving consensus is impossible [12].

Other theorems such as the Fischer-Lynch-Paterson impossibility result even show that, under conditions such as asynchronous message propagation, consensus is impossible with just a single faulty process [13]. Asynchronous message propagation means that messages do not have to arrive within a certain known time limit, and so never time-out. At first sight, this seems to be the case for Bitcoin transactions, which can propagate and be validated at any time. To make things even more complicated, the lack of identities in a peer-to-peer network opens up for Sybil attacks, in which a malicious adversary create copies of processes that from the outside look like many different nodes, when they are in fact all under the control of the malicious actor. The lack of identities makes it in general impossible to select leaders among the nodes. Many consensus protocols depend on selecting a node as leader in the network to guide in establishing consensus around some value.

But it turns out that the impossibility results were proven in very specific models, typically intended to study distributed databases. Bitcoin violates many of the assumptions implied in those models. Ironically, with the current state of research, consensus in Bitcoin probably works better in practice than in theory. Consensus can be observed as working, but the theory to fully explain why it works is not completely developed and explored yet, leading to interesting new research showing that e.g. the threshold for a 51% faulty nodes attack considered in Bitcoin to define the line after which things could become bad, eventually must be adjusted down [11]. Although not the focus of this thesis, developing theories around this part of Bitcoin is also an important field, as it can help to predict unforeseen attacks and problems.

The way Bitcoin violates assumptions of traditional models for consensus is by introducing incentives, which work specifically in the case of Bitcoin since it models a currency. Second, it makes use of randomness to remove the notion of a specific starting point and ending point for consensus used in many protocols. Bitcoin achieves consensus over a long period of time, about an hour in practice. And even at the end of that time, there is no strict guarantee that any particular block has made it permanently into the ledger and thereby established full consensus. But the probability that any block that is part of the ledger will be removed declines exponentially with time and becomes negligible after a relatively short time period. With those prospects of Bitcoin in mind, let's return to Chuck, who wants to extend Bobcoin to make it a decentralized cryptocurrecy. To understand how cryptocurrencies use Game Theory by incentivizing fair play, we will neglect all the technical details for a moment.

5.3 Making the simple crypocurrency decentralized: Version 3

Chuck is unhappy with the central role that Bob plays in Bobcoin. He wants to create a new cryptocurrency called Chuckcoin, but without being the sole authority to validate transactions or build blocks containing them. He wants to delegate this work out to everybody involved in the system. He invents a way to do so, which may sound bizarre at first, but fundamentally works in a similar way to Bitcoin.

Chuckcoin will rely on the data structure and propagation of transactions as in Alicecoin, and on a blockchain ledger as in Bobcoin, but not relay on only a single central entity that creates blocks. Every day Chuck will hold a public lottery, in which anyone can participate for free, and the winner of which receives 25 Chuckcoins as the price. Chuck will make pieces of paper that each contain a number from 1 to n, where n is the total number of participants that day. Those will be put inside a bag from which everyone takes out exactly one piece. The winner is the one that draws the smallest number, namely number 1.

The winner is allowed to create one new block that day, containing valid transactions, which typically were broadcasted during that day. Furthermore, within this block the winner may create and include one GenCoin transaction creating 25 Chuckcoins, and which are paid to himself. This GenCoin transaction serves as the reward for winning the lottery, and establishes the incentive to create a valid block, since it will then be accepted by everyone and therefore the GenCoin transaction would also implicitly be accepted.

As was also the case of Bobcoin, everybody must check the validity of this new block independently, by making sure it contains a hash pointer to the previous valid block from the previous day, that all transactions are valid and not double spends, that the GenCoin transaction does not create more coins than the allowed limit of 25 and that the whole block is signed by the rightful winner of that day's lottery.

Participants have an incentive to create valid blocks, since for a lot of times they may have traveled a long way to participate in the lottery; and when they finally win, they want to receive a reward from the GenCoin transaction that they include in their winning block.

An evil participant who for whatever reason wants to attack Chuckcoin by creating invalid blocks, could try so by e.g. double spending transactions in his block. But he would simply just lose his GenCoin reward. The block would easily be spotted as being invalid, and it would therefore immediately be rejected by all the honest participants. They could re-elect a new random winner by holding another lottery that day and hopefully culminating in an honest winner creating a valid block. The attacker could try to find ways of obtaining more votes in the lottery, to hopefully win a lot of times in a row and create a lot of invalid blocks to destroy the system. He could start to bring his friends and cousins to work for him as evil participants. But if the system gains popularity, this would probably not be enough relatives. It is difficult for the attacker to obtain enough votes for the attack to become statistically viable during longer periods of time. It would require a majority of the votes in the long run. He would therefore have to start paying people a bribe for attacking the system. But the bribe must be more than what would be obtained by just being an honest participant, who would then receive the GenCoin transaction instead.

If Chuckcoin would gain enough momentum, then each GenCoin reward could become worth so much money that the attack just becomes astronomically expensive to carry out for longewr periods of time.

Chuckcoin relies on the majority of participants being honest, which may be achievable from the incentive of playing by the rules. Using the lottery, Chuckcoin has become somewhat decentralized. Although to begin with Chuck will still be the one to organize the lottery, if one day he would just disappear, the remaining participants could still hold these lotteries without him. In the following we will see that indeed this lottery, which in the analog world sounds impossible to establish, in fact can be implemented digitally in a very elegant and completely decentralized way using hash functions.

But to assure that all participants of ChuckCoin can verify the proposed winning block, Chuckcoin still relies on people meeting in a central place to hold the lottery. It is therefore strictly speaking not peer-to-peer. Bitcoin's blockchain technology solves all of this, but introduces an obstacle in the immutability of the blockchain ledger by allowing for the negligible probability that already valid blocks in the chain can be changed for other valid blocks.

5.4 The Bitcoin Blockchain

Bitcoin makes use of a distributed ledger called Blockchain, which we have already discussed to some extent. Anybody can choose to be part of the peer-to-peer network of Bitcoin and download the entire Blockchain, a complete ledger containing every valid Bitcoin transaction made and agreed upon until today's date.

The name Blockchain is derived from the way in which new transactions are added to the ledger. Transactions are bundled together into a block that contains a reference to the previous block added. New valid transactions are added to the Blockchain every ten minuts by blocks that are added one after one, and since every block contains a reference to the previous one, they append unto an ever-growing chain of blocks.

Nodes in the Bitcoin network can choose to participate in a challenge that determines who is allowed to add a new block to the Blockchain. Bitcoin uses Game Theory to create a challenge

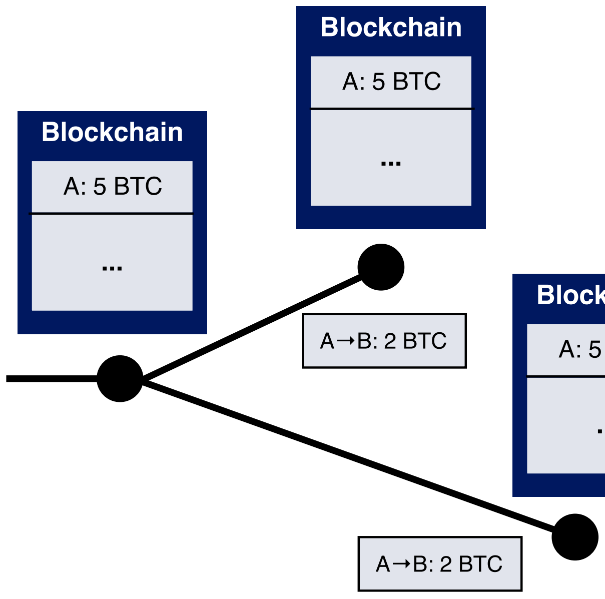


Figure 5.4.1: Simplified Blockchain representation. Every node stores the Blockchain ledger locally and forwards each new unconfirmed transaction it receives to all other nodes. In this example, one transaction is propagated that sends 2 BTC from A to B. Eventually, with the next block this unconfirmed transaction will be added to the Blockchain.

with an incentive of being an honest node in the network with a reward in bitcoins. This process is called "mining". The challenge involves the hashcash proof-of-work, which is a moderately-hard hash-puzzle, discussed in Chapter 2.

A block contains zero or more new transactions, a reference to the previous block by using a hash-pointer, a timestamp, a nonce, which is a random value, and may contain one so called coinbase transaction, which is similar to a standard Bitcoin transaction but does not need any input, thereby creating new bitcoins as a reward to the miner. When a node receives a new block from any other node it will check if the block is valid or not. For a block to be valid, all transactions contained in it must be valid with respect to all the transactions already stored in the node's local Blockchain. Therefore, a transaction in a new block cannot contain double-spends of funds or otherwise invalid transactions, and this is verifiable individually by every participating node. Furthermore, the time-stamp must be within 2 hours of the node's current time, which allows for small discrepancies in time.

The challenge of "mining" involves the hash-value of the block, which, represented as a number, must be less than an agreed-upon value by the network, called the difficulty. The challenge consists of using different nonce values within the block, until one is found that satisfies

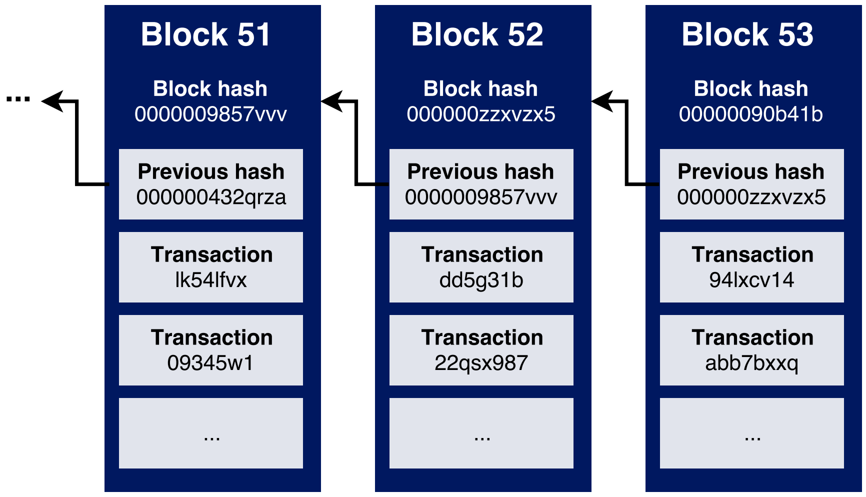


Figure 5.4.2: The Blockchain is an ever appending list of blocks, each connected through a hash pointer to its previous block. The hash of each block must satisfy a specific pattern, determined through a process called "mining".

the difficulty. If a miner finds a new block, it will propagate it to the whole network, and every node will check its validity. If the block is valid, then miners will individually add it to their own local version of the distributed Blockchain. The coinbase transaction of the new block will then implicitly be considered a valid transaction and reward the miner who found the block with bitcoins.

The miners will then try to find a new block referencing this last one and include new transactions that have not yet been confirmed. As mentioned, blocks are found in around a ten minutes interval. This is achieved by adjusting the difficulty, such that it becomes too hard for the whole network to find a block faster than ten minutes, but easy enough for it to not take longer in average. After every 2016 blocks, the whole network will adjust the difficulty, using rules defined in the Bitcoin protocol. The time it took to find the last 2016 blocks is compared to the 20.160 minutes it should have taken if every block would be found in ten minutes. The difficulty is then adjusted accordingly, with a factor corresponding to the time it took to find the last 2016 blocks divided by 20.160 minutes. In this way the Bitcoin protocol ensures that it will always take about ten minutes to add a new block regardless of the size of the network or the sophistication of the mining hardware it employs.

Since the Bitcoin software was introduced in 2009, the first block in the Blockchain is hardcoded into the software to have a united starting point, also referred to as the Genesis Block. When new nodes are added to the network, they first have to download the current Blockchain

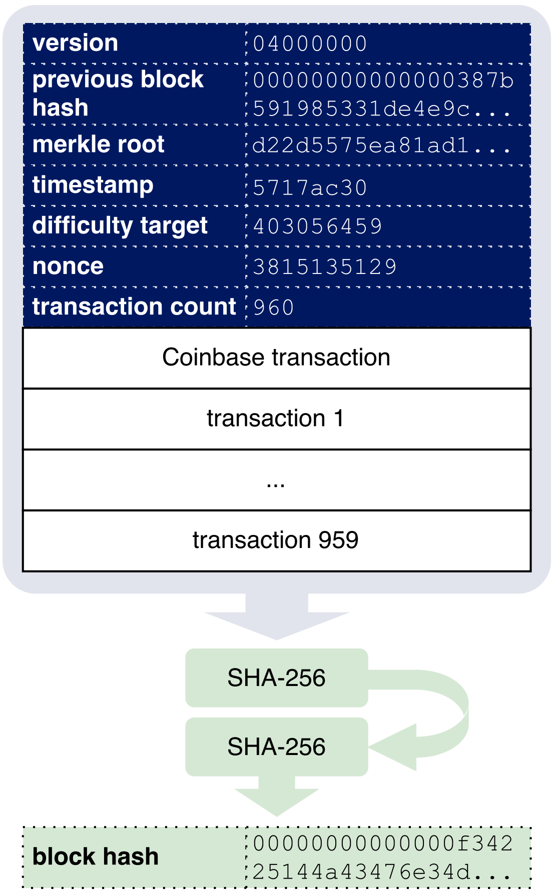


Figure 5.4.3: The data structure of a block.

by downloading it block-by-block from other nodes and validating it. When the node reaches the top of the Blockchain, it can become a mining node by starting the mining process. But as of today many different kind of mining nodes exist, some of which do not need the Blockchain, but instead trust in a network to deliver correct information to mine on.

Propagated but yet unconfirmed transactions are said to have zero confirmations. The block that includes a transaction counts as its first confirmation, because the whole network has accepted and agreed to view it as a valid transaction and can therefore not be double-spent in this Blockchain anymore. Every block mined thereafter counts as further confirmation. When the next block is mined, it will refer to the block containing the transaction as its parent block and will count as the second confirmation for that transaction. The next block will refer to the previous block and count as a third confirmation, and so forth. Every new-mined block digs the transaction further and further into the Blockchain.

The rule for creating a coinbase transaction is specified by the Bitcoin protocol. The maximum value allowed in a coinbase transaction was 50 bitcoins to begin with, and the reward is halved after every after 210.000 mined blocks . Although the difficulty is adjusted such that it should take approximately four years to mine those 210.000 blocks, in practice the network has been growing so fast, that the last halving only took three years to occur. The first reward halving occurred in 2013, and miners could from then on only reward themselves with 25 bitcoins in each block, expected to halve again in July of 2016. This is the only way bitcoins can be created, and hence all bitcoin in circulation can be traced back to such a coinbase transaction. Furthermore, each Bitcoin transaction must contain a small fee, depending on the size of the transaction. The fee is the difference of the amount in bitcoins from the inputs and the amount of bitcoins send with the outputs. The sum of all these fees from transactions included in a block can be added to the coinbase transaction as additional reward for the miner. Thus when no more new bitcoins are generated, an incentive of mining blocks contiues to come from the reward of the transaction fees. A bitcoin is dividable into 100.000.000 subunits, which are called Satoshis in honour to the creator of Bitcoin. As a result, since 32 subsequent reward halvings would require smaller subunits than one Satoshi as a reward, no more bitcoins will be created after around 130 years. The total amount of bitcoins to be created is therefore capped at roughly 21 million, or $2.1 * 10^{15}$ Satoshis, which gives Bitcoin its property of scarcity.

The Bitcoin protocol dictates to always use the Blockchain with the most cumulative difficulty as the one to establish consensus around. This means that every node with different Blockchains to choose between, will choose to mine blocks for the Blockchain where the sum

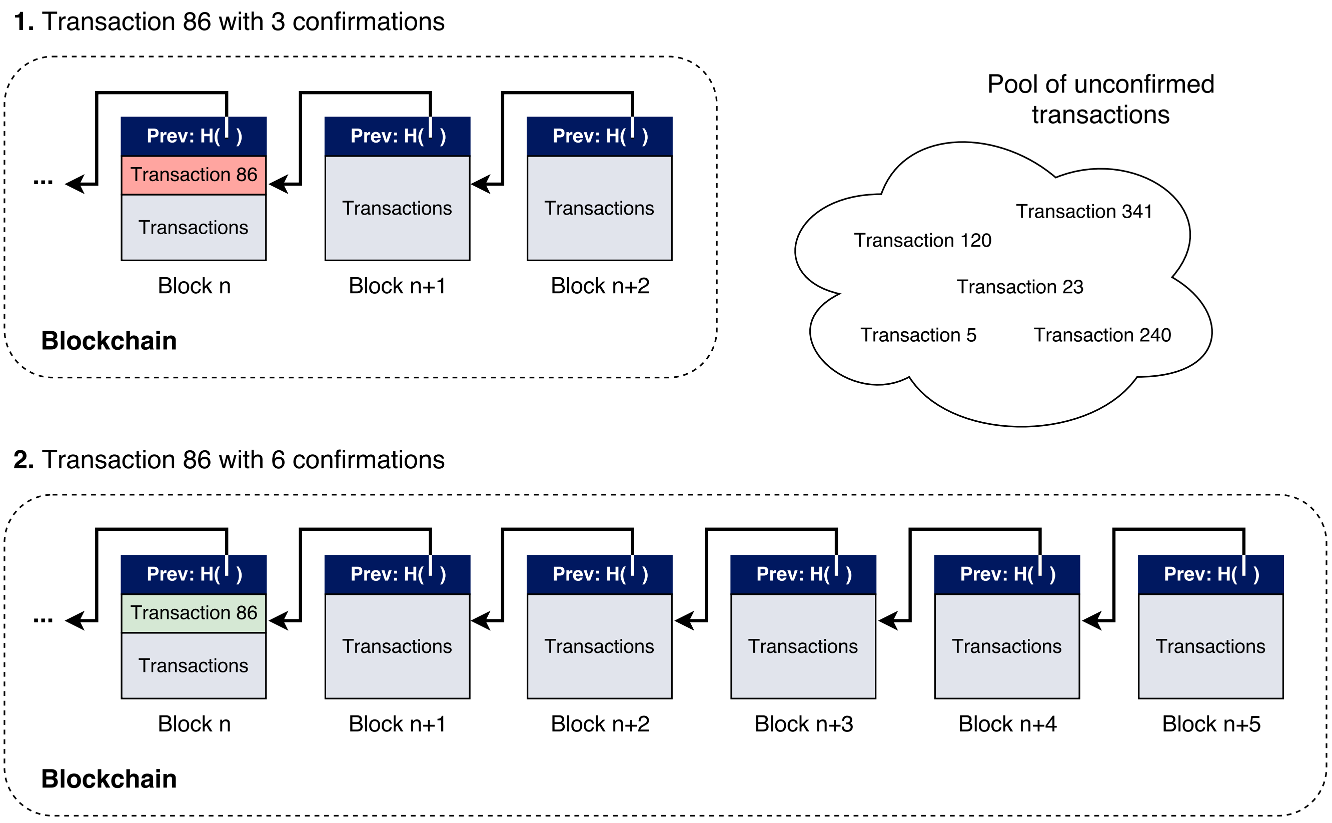


Figure 5.4.4: Showing a transaction with two and six confirmations. The pool of unconfirmed transactions have propagated through the network but are not yet included in any block in the Blockchain. As a rule-of-thumb, a transaction is considered *confirmed*, if it has at least six confirmations (indicated with green).

of difficulties from all the blocks of that Blockchain is the greatest. This is typically also the longest Blockchain. It prevents the Bitcoin network from splitting permanently into two or more separate networks, working on two different Blockchains, also referred to as a Blockchain fork. It can happen and does so quite often for a short period of time. When two miners independently find a new block at nearly the same time, one part of the network could receive one block at first and accept it, while the other part receives the other one first and accepts that one. Now the network is working with two slightly different Blockchains, being equal up to the block before the last one, but differing on the last block. Their cumulative difficulty is the same, so they will choose to work on the Blockchain from which they received the last block first. But if in either one or the other part of the network split a new block is found, it will be propagated to the whole network. THerefore, nodes working on the other Blockchain will also receive the new block and see that the other Blockchain now contains more cumulative difficulty, namely is one block ahead. They then switch to the other. This network split has thus resolved itself, and the neglected branch of the Blockchain will be referred to as orphaned. The whole network is therefore self-healing and now in consensus about exactly one Blockchain again. But for that reason, a transaction that has just been validated in a block is not quite as likely to stay in the Blockchain that the network establishes consensus around as one that has already hundreds of confirmations.

If there is a bad actor with a lot of hashing power to mine blocks faster than the rest of the network, he could reverse a newly confirmed transaction in the following way. The attacker could exploit Bitcoin's protocol rule by first propagating a transaction of his own to the network. When the transaction is confirmed in a block, the attacker will not try to mine a new block that refers to that last found block, which confirmed his transaction. Instead, he will try to find a new block with a reference to the next-last block, which is the parent block to the one that confirmed his transaction. This block could then contain a transaction, which would be a double-spent of his already confirmed transaction, but still be a valid block, since it is in a fork which does not contain his first transaction that already spent the funds. If the attacker propagates such a block to the network, then every node will accept it as a valid block and store it as a parallel chain. But they will not change to this parallel chain, since it is of the same cumulative difficulty, and its last block was not received first.

The attacker can only succeed with his attack if he could now find another block that builds on his last one, before the rest of the network finds a new block to their version of the Blockchain. Then the network would change to the attacker's Blockchain, which now contains more cumula-

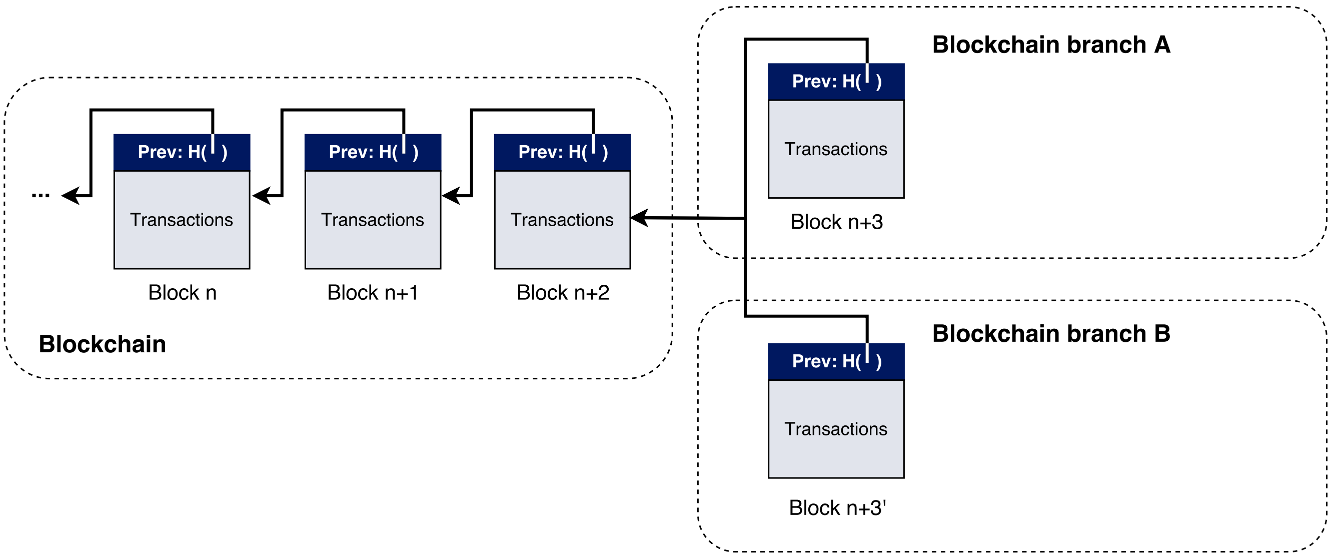


Figure 5.4.5: The Blockchain has branched and one part is working on branch A and the other on branch B. This can happen when block n+3 and n+3' are found at nearly the same time, and one part of the network receives block n+3 and the other block n+3' first.

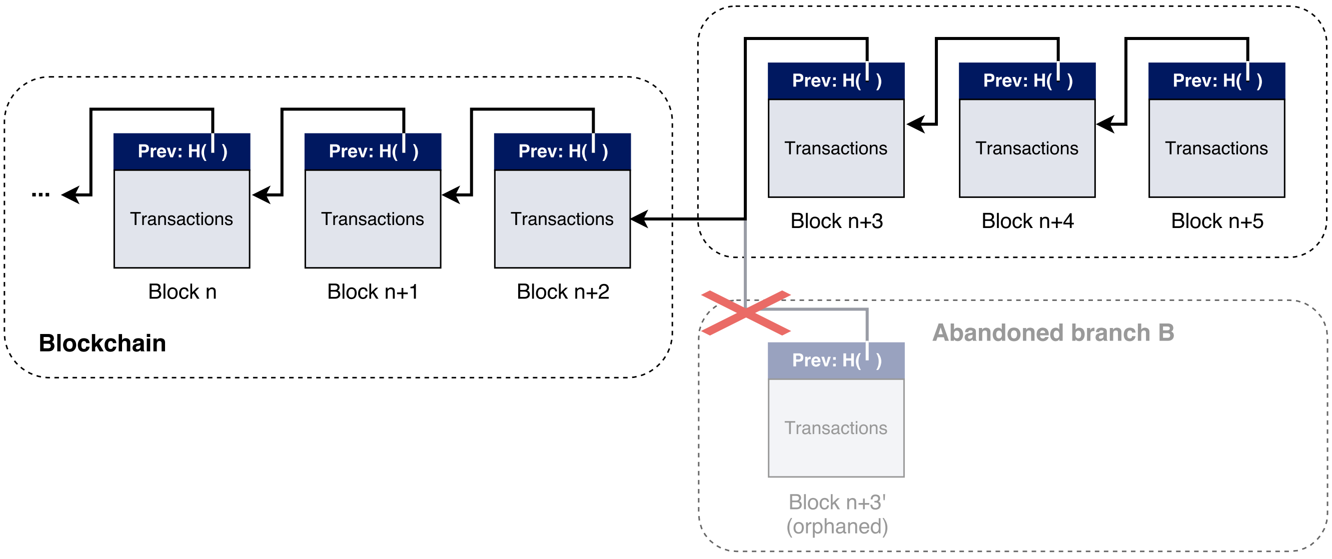


Figure 5.4.6: The network has orphaned branch B by continuing with only the branch of highest cumulative difficulty.

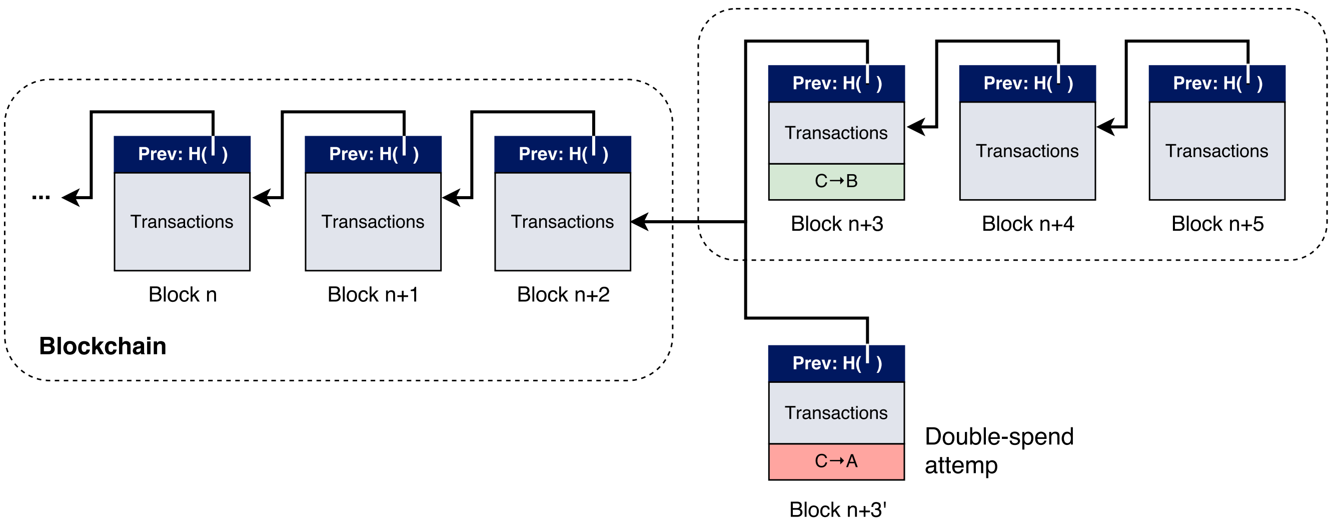


Figure 5.4.7: A double spend attempt. The attacker needs to create a block referencing the one before his transaction he wants to double-spend was confirmed and continue creating new blocks until he outperforms the current Blockchain.

tive difficulty. This would successfully change the consensus of the network from first agreeing upon a transaction with one confirmation, but then removing it from the Blockchain and accept the double-spent of that transaction instead. Nevertheless, an attacker could only double-spent his own transactions, or choose to exclude transactions in his own blocks. He could not change balances or destination addresses of transactions, since they are cryptically secured as described in the previous chapter using public-private key encryption and signatures. Since the attacker needs a lot of hashing-power, it becomes very expensive and difficult to achieve for even small time periods. For transactions with many confirmations, an attacker would have to create a parallel Blockchain starting from a block deep inside the chain where the transaction was first confirmed, and mining up until the point that this parallel chain has more cumulative difficulty than the "original". This is statistically only probable to happen when the attacker controls more than 50% of the network's hashing power. This kind of attack is therefore called a 51% attack. The probability to achieve this with less than 51% of the hashing power and a transaction that has 6 confirmations, is in most cases negligible. As a rule of thumb, after six confirmations a transaction is considered to stay in the Blockchain forever and therefore mostly safe to accept as a payment even when it contains the transfer of value worth millions of dollars. As of February 2016 the total hashing-power of the Bitcoin network was over 1 exa hashes per second, which corresponds to more than 10.000 times of what the computing power of the top 1000 known super-computers combined can deliver. Bitcoin is by far the world's biggest distributed computing system. Although a transaction can be created using unspent transactions as inputs, it can of course only be confirmed when all of its inputs are confirmed in the Blockchain, or when they do all get confirmed with the same block. A block whose inputs are yet unconfirmed would result in an invalid block. To prevent a coinbase transaction from being spend too early after confirmation, raising the possibility of later invalidating the coinbase transaction due to a small Blockchain fork, the protocol specifies that a coinbase transaction must reach "maturity" before it can be spent. This means that the transaction must have at least 144 confirmations before being spend, which corresponds to roughly one day.

5.5 Changing consensus rules

Bitcoin has a lot of deliberate constraints hard-coded into the protocol, which were chosen when Bitcoin was proposed in 2009. Those are among others the aim on the average time per block, the size of blocks, and the divisibility of the currency, the total number of Bitcoins, and the block reward amount.

The limitation of the supply of currency available will likely never change, as there was from the beginning of the proposal a clear vision that this is the one rule that should never be changed, no matter if the specific constrain was wisely chosen or not.

Other choices, such as the block size limit or the available opcodes, may occasionally be changed. Proposals for changing the protocol, adding feature or just information are specified in so called Bitcoin Improvements Proposals (BIP). To put a BIP to practive in Bitcoin, often requires the consensus rules in the protocol to change. Due to the decentralized protocol, it is not as easy as just updating the rules, releasing the new software, and expecting everybody to instantly run this new version. The consequences of having some nodes run outdated software, depend on the changes the new version would bring. There is a clear distinction between those consequence, which can be categorized into *soft forks* and *hard forks*.

A soft fork is a change that makes validation rules stricter. It adds features that restrict the set of valid transactions or blocks, such that the new version would reject some of the blocks that the previous version would accept. As long as the majority of the network switches to the new version, this would avoid a permanent split in the network to happen. The nodes running this new version can then enforce their stricter rules, while the nodes running the previous version would continue to mine on the same blockchain, since valid blocks in the new version are also valid in the previous version. But eventually a node running the previous version would mine what is considered an invalid block by the new version. But it would be rejected by the majority of the network, since it is running the new version. Nodes running the previous version would therefore need to update their software to not introduce a handful of unused and orphaned blocks. A so called Pay-To-Script-Hash transaction type which we will meet in Chapter 6 is one example introduced as a soft fork.

A few months before the time of this writing, there is a new interesting BIP [26], which could make all BIP proposals made afterwards implementable by a soft forks. This proposal will briefly be discussed in Chapter 9.

A hard fork on the other hand is a change that introduces features previously considered invalid. The new version would then mine blocks that the previous version will not accept. Even after the majority of nodes upgrade to the new version, the remaining nodes running the previous version will regard the blockchain branch that the new version uses as invalid, and continue to mine on their own branch. Hence the network will split and produce two parallel branches of the blockchain. Those two branches could never be joined together again, which is considered unacceptable and avoided in most cases.

Until today a hard fork has been tried to realise on the Bitcoin network. But a voting mechanism, in which miners can express their willingness to update to a new version before it would activate, would probably mitigate serious problems from a hard fork. Such a voting mechanism is already in use and implemented by encoding small messages inside blocks, expressing the miners voting favour.

Part II

A programming model

Bitcoin may be the TCP/IP of money.

Paul Buchheit

6 The Script language

Script is a programming language and the core of Bitcoin transaction processing. It resembles the programming of assembly code. A Script program will execute and return true if it was successful.

Script does not allow looping, and Script programs always terminate. The memory in Script is accessed through a stack, in which the last item pushed onto is the first item to be popped out. There is therefore no such thing as variable names in Script. Calculations and data manipulation are just done directly on the stack. Typically, the stack items become operands of subsequent opcodes. When the script terminates, the top stack item is the return value.

6.1 A model for Script

Recall that any transaction output is always completely consumed by a spending transaction. Therefore transactions form chains of consumed outputs up to the last one, which is still unspent. Every output of a transaction contains one part of a little program written in Script, which was previously referred to as a lock-script. It specifies conditions that must be met to spend bitcoins from an output. In technical terms the lock-script is also called a *scriptPubKey*, a name it has obtained from the white paper that specifies the Bitcoin protocol [18].

Inputs to transactions contain an unlocking-script, referred to as a *scriptSig* in technical terms. It must satisfy the conditions required from the scriptPubKey of the output it spends. Although the scriptSig can be a Script program, there is no need for it to calculate any values, since it is evaluated without the possibility of receiving any inputs. Therefore all values can be just directly declared in the scriptSig.

Context

The Blockchain was introduced in Chapter 5 and contains the history of all transactions ever made and agreed upon in Bitcoin. But for verifying transactions, the history is actually not so important. Instead, it is the currently unspent transactions which matter. Due to the Blockchain, at any given time the network has implicitly also agreed upon a set of valid unspent transaction outputs (*UTXO*). The set of UTXO contains the transaction hashes, acting as unique identifiers, output indices, output amounts and scriptPubKey for all the unspent transaction outputs. This set therefore contains the only possible unspent transaction outputs from which any new transaction or chain of transaction must spend.

In the model, the set of UTXO is accessible in the context through a function C, which as input takes a transaction hash, an output index and a parameter, and returns either the output amount or scriptPubKey, depending on the parameter.

More specifically,
C is a function that returns an unspent transaction output from the set of UTXO.
$C(h, n, 0) = v$, where h is a transaction hash, n is the output index and v is the bitcoin amount for this specific transaction output.
$C(h, n, 1) = P$, where P is the scriptPubKey for this specific output.

Transactions

Transactions are simply formalized as a composition of inputs and outputs, defining a pair. The transactions of the model therefore intuitively resemble Bitcoin transactions.

A transaction t can be defined as follows:

$$\begin{array}{lcll} t & = & (\widetilde{I}, \widetilde{O}, l), \text{ where} & \\ \widetilde{I} & = & (I_1, \ldots, I_n), & \text{n : \# of inputs in t} \\ \widetilde{O} & = & (O_1, \ldots, O_m), & \text{m : \# of outputs in t} \\ l & = & \text{is the locktime of the transaction} & \end{array}$$

I_i is the i-th input of transaction t, and therefore has input index i.
Thus,
$I_i = (h_i, n_i, S_i, seq_i)$, where S_i is the scriptSig satisfying the scriptPubKey from the transaction with hash h_i and output index n_i. The value seq_i specifies the sequence number of this input. As mentioned in Chapter 4, it is assumed that the sequence number is regarded as a relative locktime specified in the Bitcoin proposol BIP68 [14].

O_j is the j-th output of transaction t, and therefore has output index j.
Thus,
$O_j = (P_j, v_j)$, where v_j is some bitcoin amount being sent to this output, and P_j is the scriptPubKey that locks those bitcoins.

Updating the context

Unspent transactions that are added to the blockchain must also be added to the set of unspent transaction outputs (UTXO). At the same time, the outputs from which those transactions spend must be removed from the set of UTXO.

In the following, v_i denotes the the amount contained in the output that the i-th input is referencing, and v_j is the amount locked to the j-th output of transaction t. Furthermore, P_i is the scriptPubKey of the transaction output refernced by the i'th input of transaction t. Concatenating the scriptSig S_i with the scriptPubKey P_i yields what will be called a *Script program*.

A transaction is valid if: for all inputs the corresponding Script programs are valid; the total amount of bitcoins send with the transaction outputs do not exceed the total amount referenced with the inputs; the locktime of the transaction is not in the future; and the sequence number for each input has reached "maturity". To understand the use of locktime and sequence number, recall their definition from Section 4.2. Specific details will follow in this chapter. To decide if a transaction must update the context, exactly these conditions must be met.

$$\frac{\forall I_i \in \widetilde{I} :\ S_i.C(h_i, n_i, 1) \xrightarrow{*} \texttt{valid} \qquad \sum_{I_i \in \widetilde{I}} C(h_i, n_i, 0) \geq \sum_{O_j \in \widetilde{O}} v_j \qquad past(l) \qquad mature(seq_i)}{C \triangleright (\widetilde{I}, \widetilde{O}, l) \ \rightarrow \ C'}$$

$$\text{where } C' = C \quad \begin{aligned} &-[\,(h_i, n_i, k) \uparrow \ \mid I_i \in \widetilde{I}, k \in \{0,1\}\,] \\ &+[\,(h(t), n_j, 0) \mapsto v_j,\ (h(t), n_j, 1) \mapsto P_j \mid O_j \in \widetilde{O}\,] \end{aligned}$$

If $past(l)$ is true, it simply means that the locktime of the transaction is set to a date, or block number, that is not in the future. For $mature(seq_i)$ to be true, the output that input I_i spends must have been added to the UTXO for a timespan of at least the one that seq_i defines. The details regarding the notion of time in the context are omitted, but could basically just be added as an additional structure. Further technical rules defining the maximum size of transactions and programs are left out of the model of the context, but will be mentioned when applicable later.

In the resulting context, the the spent transaction outputs are removed, while the outputs of transaction t are added to C, where $h(t)$ is the transaction hash.

Technically, transactions updating the context are provided by new blocks to the blockchain. Transactions are not explicitly ordered within a block. But since an input always spends exactly one output, it is trivial to rearrange them into chains where necessary, and then by starting from the tail of each chain, checking that the transactions are valid and applying the update of the context for each transaction.

Script

Script is a simple, stack-based and purposely Turing incomplete language without loops. It is essentially a list of instructions, informally referred to as *Script words*, but technically called *opcodes*. As we already discussed, a Script program always consists of a scriptSig with a scriptPubKey appended to it.

Recall that a typical transaction transfers bitcoins with its outputs to a destination address D, simply by encumbering future spending of the bitcoins with two things the spender must provide together with the scriptSig of a spending traansction:

- a public key that, when hashed, yields the destination address D embedded in the scriptPubKey, and

- a signature to show evidence of the private key for the public key just provided.

These instructions are provided by programming the scriptPubKey using Script. But much more complex Script programs are possible. The Script language provides flexibility to change the parameters of what is required to spend the transferred bitcoins. The scriptPubKey could for example be programmed to require two private keys, or a combination of several, or even no keys at all.

Anyone-Can-Pay

Lets take a look at the simplest possible Bitcoin Script:

```
scriptPubKey:  (empty)
scriptSig:     OP_TRUE
```

The scriptPubKey is empty, so there is no specific condition that must be met to spend the funds from a transaction output with this scriptPubKey. However, one general condition must always be met. A Script program must terminate with the value `true` left as top element on its stack. Otherwise, the whole transaction is marked as invalid. For our empty scriptPubKey above, the scriptSig therefore suffice to use the opcode `OP_TRUE`, which simply pushes the value true to the stack. In fact, since `true` is represented by any value different from zero, also any other operation pushing some value different from 0 could have been used instead. Details regarding the representation of true and false will be mentioned later for "Stacks and values". After that instruction, the program reaches its end and will be considered valid.

A transaction output containing this kind of scriptPubKey would be spendable by anyone, and is therefore also called an Anyone-Can-Pay transaction. But in particular the miner who creates the block that includes this transaction can spend it. He could spend the output immediately within the same block it is added to, by creating a transaction that references those funds in its input, and provides the scriptSig from above. The scriptPubKey in the output of his transaction should then contain the typical instructions that lock the funds to his own Bitcoin address.

Pay-To-PubKey

Let's now take a look at a more useful Script program, which is a type of transaction called a *Pay-To-PubKey*. The name scriptPubKey and scriptSig originate from this type of transaction:

Table 6.1.1: The execution of a Pay-To-PubKey transaction type

Stack	Script	Description
`(empty)`	`<sig> <pubKey> OP_CHECKSIG`	The *scriptSig* and *scriptPubKey* is combined.
`<sig>`	`<pubKey> OP_CHECKSIG`	The constant `<sig>` is pushed to the stack.
`<sig> <pubKey>`	`OP_CHECKSIG`	The constant `<pubKey>` is pushed to the stack.
`true`	`(empty)`	Signature is checked for top two stack items and evaluates to true if correct.

```
scriptPubKey:  <pubKey> OP_CHECKSIG
scriptSig:     <sig>
```

The constant "`pubKey`" refers to a public key of a ECDSA key-pair and "`sig`" to a signature for that public key. When executed, they will simply be pushed onto the stack, which is indicated by using angle brackets ("`< >`") around them. We will later discuss more details regarding the opcodes that push data. The next instruction in the program to consider is the opcode `OP_CHECKSIG`. It involves a lot of steps to check that the provided signature is for the spending transaction and provided public key. It will also be thoroughly discussed later.

The Script program, consisting of combining the scriptSig and scriptPubKey from our example above, is then `<sig> <pubKey> OP_CHECKSIG`. The execution is from left to right and can be seen in Table 6.1.1.

The Pay-To-PubKey transaction type was used by early versions of the Bitcoin protocol and is still supported by the network. Its disadvantage is that it involves long public keys in the output of transactions. It has been outdated by another transaction type called *Pay-To-PubKey-Hash*.

Pay-To-PubKey-Hash

Instead of using a public key directly, a Pay-to-PubKey-Hash transaction uses a Bitcoin address. As described in Section 4.3, there is a direct correspondence between a public key and a Bitcoin address, since the Bitcoin address consists of the hashed value of the public key. The scriptPubKey of a Pay-To-PubKey-Hash only incorporates this hashed value, which is smaller than the public key. The public key should then instead be provided by the scriptSig. The advantage is that it reduces the size of the UTXO, which contains all the unconfirmed transaction outputs and therefore also the scriptPubKeys. But it increases the overall amount of data stored in the blockchain. This transaction type integrates well with the notion of value transfer. There is only the need of exchanging Bitcoin addresses, which are shorter than public keys and whose accuracy can be checked with the checksum inside the address. The scriptSig containing the public key is provided by the receiver of the funds, and he will as a matter of course know his own public key.

The Script program is similar to the Pay-To-PubKey above, but with the additional step of checking that the provided public key corresponds to the hash value in the scriptPubKey:

```
scriptPubKey:  OP_DUP OP_HASH160 <pubHash>
               OP_EQUALVERIFY OP_CHECKSIG
scriptSig:     <sig> <pubKey>
```

The execution is shown in Table 6.1.2.

Pay-To-Script-Hash

To simplify the use of complex transaction scripts greatly, yet another type of transaction called a *Pay-to-Script-Hash* (P2SH) was standardized in the BIP16 proposal in 2012 [1]. It allows transactions to be sent to the hash value of a script instead of the scriptPubKey itself. The script hash can be encoded similar to a Bitcoin address. To spend bitcoins sent as a P2SH, the recipient must provide a script matching the script hash, and data that makes this script valid. The scriptPubKey is defined as:

```
scriptPubKey:  OP_HASH160 <20-byte-hash-value> OP_EQUAL
```

A first sight, this scriptPubKey may look too simple. Before the P2SH transaction type was defined, this output could have been simply spent by providing a hex string, whose hash corresponds to `20-byte-hash-value` referenced in the scriptPubKey. But BIP16 introduced a special rule regarding exactly this scriptPubKey, recognized by its pattern, and not directly enforced through the opcodes.

The P2SH transaction locks the output to the hash value of what it expects to be a script. It is equivalent to saying "pay to a script with this hash", and the sender only needs to know the much shorter and simpler hash value. The script whose hash was locked in the output is then referred to as the *redeem script*. When the funds must be spend, this redeem script must be presented in the scriptSig, together with values that satisfy it. The redeem script is presented

Table 6.1.2: The execution of a Pay-To-PubKey-Hash transaction type.

Stack	Script	Description
`(empty)`	`<sig> <pubKey> OP_DUP OP_HASH160 <pubHash> OP_EQUALVERIFY OP_CHECKSIG`	The *scriptSig* and *scriptPubKey* is combined.
`<sig> <pubKey>`	`OP_DUP OP_HASH160 <pubHash> OP_EQUALVERIFY OP_CHECKSIG`	Constants are added to the stack.
`<sig> <pubKey> <pubKey>`	`OP_HASH160 <pubHash> OP_EQUALVERIFY OP_CHECKSIG`	Top stack item is duplicated.
`<sig> <pubKey> <pubKeyHash>`	`<pubHash> OP_EQUALVERIFY OP_CHECKSIG.`	Top stack item is hashed.
`<sig> <pubKey> <pubKeyHash> <pubHash>`	`OP_EQUALVERIFY OP_CHECKSIG.`	Constant added.
`<sig> <pubKey>`	`OP_CHECKSIG`	Equality is checked between the top two stack items.
`true`	`(empty)`	Signature is checked for top two stack items and evaluates to true if correct.

in the scriptSig through a single push that pushes the whole redeem script in its serialized form onto the stack. The P2SH script is then executed in two stages.

First, the redeem script that is present on the stack is checked against the hash value from the scriptPubKey, just as the instructions would suggest. But if the hash matches, then also the whole scriptSig including the redeem script is executed on its own and must of course also evaluate to true.

Using P2SH, bitcoins can be send to addresses secured in various unusual ways, without knowing about the details of how the security is set up in a script. P2SH has been typically used to make it much easier to pay to a so called multisignature output, in which a number of signatures from different public keys can be required. With P2SH a sender only needs to know the corresponding script address, instead of knowing all the public keys that are part of the multisignature. It is then the burden of the recipient to provide the multisignature script when spending the funds.

Just like P2PKH, also P2SH shifts the burden in data storage for a long script from the output, that is in the set of UTXO, to the input, stored solely in the blockchain.

Stacks and values

We have seen that Script uses one stack, onto which values can be pushed and popped during execution. The stacks holds byte vectors. When represented as numbers, byte vectors are interpreted as little-endian variable-length integers with the most significant bit determining the sign of the integers. This will from now on just be denoted as 64 bit signed integers. The value for `false` is zero or negative zero, using any number of bytes, or an empty array, and `true` is anything else.

For our formal model of Bitcoin transactions, we will specify Script with two stacks, each of which have a different purpose. The stack we saw being used in the examples above was the value stack, which we will denote as $\widetilde{v}$. Furthermore:
$\widetilde{B}$ is the Boolean execution stack to keep track of the flow control. Initially it is empty, and when a Script program terminates, it must be empty.
P is a Program written in Script.

At termination, a transaction will be valid if `true`, which is any value different from 0, is on the top of the value stack $\widetilde{v}$. Furtermore, the Boolean execution stack $\widetilde{B}$ must be empty at termination.

6.2 Script words – commands and functions

The labeled names of instructions in a Script program are informally called Script words. They can be regarded as commands and functions, and in technical terms are just referred to as opcodes. There are some opcodes which existed in very early versions of Bitcoin but were removed. From time to time new functionality to opcodes are added by means of carefully designed and executed soft forks (see Section 5.5). This is done using e.g. specially reserved opcodes labeled `OP_NOP1` to `OP_NOP10`, which before such update is applied have no instruction assigned to them, and are simply skipped when encountered.

The labeled names of opcodes are not directly expressed as such in Script programs. They instead appear with a unique 1 byte hexadecimal value, which is assigned to each of the opcodes. Not all possible 1 byte hexadecimal values represent an opcode. A Script program will terminate and mark the transaction as invalid if an unassigned hexadecimal value is being evaluated as an opcode. In the following line and when applicable, we will always refer to the opcodes using their assigned labeled names, which is of course much easier than remembering their assigned hexadecimal value. Just keep in mind that the names are only representative.

In the following, a few opcodes from different categories are presented together with their hexadecimal representation in squared brackets and their semantics in our model. The rest of the opcodes are left for the appendix. Opcodes comparing values or manipulating them, with the exception of `OP_SIZE`, `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY`, pop those values and only leave the result on the stack. This will be mentioned more specifically when we will encounter those opcodes. But it will also be clear from the execution rules below.

Constants

The following opcodes push constants to the stack, without manipulating any values. For now we will only consider the value stack $\widetilde{v}$ and first introduce the Boolean stack $\widetilde{B}$ when we look at opcodes regarding the flow control.

`OP_1NEGATE [0x4f]`: Pushes -1 onto the stack.

$$\frac{}{\widetilde{v},\ OP\_1NEGATE.P\ \rightarrow\ \widetilde{v} :: -1,\ P}\ \mathrm{OP-1NEGATE}$$

`OP_0 [0x00]`: Pushes 0 onto the stack.

$$\frac{}{\widetilde{v},\ OP\_0.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \mathrm{OP-0}$$

`OP_1 [0x51]`: Pushes 1 onto the stack.

$$\frac{}{\widetilde{v},\ OP\_1.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \mathrm{OP-1}$$

`OP_2 [0x52]`: Pushes 2 onto the stack.

$$\frac{}{\widetilde{v},\ OP\_2.P\ \rightarrow\ \widetilde{v} :: 2,\ P}\ \mathrm{OP-2} : \text{Pushes 2 onto the stack}$$

...

`OP_16 [0x60]`: Pushes 16 onto the stack.

$$\frac{}{\widetilde{v},\ OP\_16.P\ \rightarrow\ \widetilde{v} :: 16,\ P}\ \mathrm{OP-16}$$

There are numerous ways to push any arbitrary bytes of data onto the stack. By the current consensus rule of the Bitcoin protocol, the maximum push of data allowed in a Script program is 520 bytes. Any transaction containing a script with a greater push than 520 bytes of data will simply be rejected by the network and is considered invalid. The limitation is however not a technical one. There exist opcodes that could potentially push data of up to 4GB of data. Without the need of changing the Script language, the 520 bytes limit could hence be lifted with an update to the consensus rules, but which would require a hard fork.

There are unlabeled opcodes that push data of 1 to 75 bytes onto the stack, which follow after the call to the opcode. The opcodes are assigned to hexadecimal values equal to the size of data they push, from `[0x01]` to `[0x4b]`.

`[0x01]`: Pushes the following $0x01 = 1$ byte onto the stack.

$$\frac{d \text{ is } 1 \text{ byte}}{\widetilde{v},\ [0x01].d.P\ \rightarrow\ \widetilde{v} :: d,\ P}\ \mathrm{OP-PUSH1}$$

$$\frac{d \text{ is} 1 \text{ byte}}{\widetilde{v},\ [0x01]\ \rightarrow\ \emptyset,\ \text{invalid tx}}\ \mathrm{OP-PUSH1}$$

Note that d is the next 1 byte in the Script program, following the call to $[0x01]$. Similarly, $[0x02]$ pushes the next 2 bytes onto the stack, and so on. If the push is the last opcode, hence there is no data to push, the transaction is marked invalid.

If data to be pushed contains between 76 and 520 bytes, then opcodes with label names `OP_PUSHDATA1`, `OP_PUSHDATA2` and `OP_PUSHDATA4` must be used. When they are called, then the next 1, 2 or 4 bytes of data contains the number of bytes it will push onto the stack, respectively.

`OP_PUSHDATA1 [0x4c]`: The next byte contains the number of bytes to push.

$$\frac{n \text{ is 1 byte} \quad d \text{ is } n \text{ bytes}}{\widetilde{v},\ OP\_PUSHDATA1.n.d.P\ \rightarrow\ \widetilde{v} :: d,\ P}\ \text{OP} - \text{PUSHDATA1}$$

$$\frac{n.d < n + 1 \text{ bytes in size}}{\widetilde{v},\ OP\_PUSHDATA1.n.d\ \rightarrow\ \emptyset,\ \text{invalid tx}}\ \text{OP} - \text{PUSHDATA1}$$

Note that n is the 1 byte containing the number of bytes to push onto the stack, and d is the data to be pushed. Since n is only 1 byte, the maximum bytes of data OP_PUSHDATA1 can push is 255 bytes. To push 256 to 520 bytes, the opcode OP_PUSHDATA2 can be used. Finally, OP_PUSHDATA4 does not add any further functionality, since it is not allowed to push more than 520 bytes currently. But it may still be used to push any allowed number of bytes. Using `OP_PUSHDATA4` always results in a bigger program size, since it requires four bytes to denote how many bytes it will push, instead of e.g. only one or two.

In the example we have already seen the simplified notation of "`< >`" is used to denote the push of data inside the brackets. This is a small-hand notation for the minimal size data push using the smallest possible opcodes. It is always favourable to push data in its most compact form, using as few total bytes as possible.

The last opcode regarding constants is `OP_SIZE`. Note, that the opcode `OP_SIZE [0x82]` does not pop its operand, but pushes the string length of the top element of the stack.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_SIZE.P\ \rightarrow\ \widetilde{v} :: v_n :: |v_n|,\ P}\ \text{OP} - \text{SIZE}$$

Stack manipulation

The following opcodes manipulate the value stack. Their execution should be clear from the specification of the rules below.

`OP_DROP [0x75]`: Removes the top stack item.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_DROP.P\ \rightarrow\ \widetilde{v},\ P}\ \mathrm{OP-DROP}$$

`OP_DUP [0x76]`: Duplicates the top stack item.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_DUP.P\ \rightarrow\ \widetilde{v} :: v_n :: v_n,\ P}\ \mathrm{OP-DUP}$$

`OP_IFDUP [0x73]`: If the top stack value is not 0, duplicate it.

$$\frac{v_n \neq 0}{\widetilde{v} :: v_n,\ OP\_IFDUP.P\ \rightarrow\ \widetilde{v} :: v_n :: v_n,\ P}\ \mathrm{OP-IFDUP1}$$

$$\frac{v_n = 0}{\widetilde{v} :: v_n,\ OP\_IFDUP.P\ \rightarrow\ \widetilde{v} :: v_n,\ P}\ \mathrm{OP-IFDUP0}$$

`OP_SWAP [0x7c]`: The top two items on the stack are swapped.

$$\frac{}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_SWAP.P\ \rightarrow\ \widetilde{v} :: v_n :: v_{n-1},\ P}\ \mathrm{OP-SWAP}$$

Logic

There are also opcodes performing logical operations. The most important is `OP_EQUAL [0x87]`. It returns 1 if the inputs are exactly equal, 0 otherwise.

$$\frac{v_{n-1} = v_n}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_EQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \mathrm{OP-EQUAL1}$$

$$\frac{v_{n-1} \neq v_n}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_EQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \mathrm{OP-EQUAL0}$$

Arithmetic

Script has a number of opcodes involving arithmetic functions. Those arithmetic opcodes do not interpret numbers bitwise, but evaluate them numerically, so that e.g. $0x01 = 0x0001 = 0x000001$. Numbers are limited to signed 8 byte integers, but when used as operands they are not allowed to be longer than 4 bytes, otherwise the transaction is immerdiately marked as invalid. However, the result of an arithmetic operation is allowed to overflow!

For the following arithmetic opcodes we assume that $v_{n-1} \downarrow n_1$, $v_n \downarrow n_2$ and $v' \downarrow n_3$, all of which are 4 byte signed integers.

`OP_1ADD [0x8b]`: 1 is added to the input.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_1ADD.P\ \rightarrow\ \widetilde{v} :: v_n + 1,\ P}\ \text{OP} - \text{1ADD}$$

`OP_SUB [0x94]`: *b* is subtracted from *a*.

$$\frac{v_{n-1} - v_n = v'}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_SUB.P\ \rightarrow\ \widetilde{v} :: v',\ P}\ \text{OP} - \text{SUB}$$

`OP_BOOLAND [0x9a]`: If both *a* and *b* are not 0, the output is 1. Otherwise 0.

$$\frac{v_{n-1} \neq 0 \land v_n \neq 0}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_BOOLAND.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP} - \text{BOOLAND1}$$

$$\frac{v_{n-1} = 0 \lor v_n = 0}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_BOOLAND.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP} - \text{BOOLAND0}$$

`OP_NUMEQUAL [0x9c]`: Returns 1 if the numbers are equal, 0 otherwise.

$$\frac{v_{n-1} = v_n}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_NUMEQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP} - \text{NUMEQUAL1}$$

$$\frac{v_{n-1} \neq v_n}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_NUMEQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP} - \text{NUMEQUAL0}$$

Cryptographic functions

Script offers a lot of cryptographic opcodes. A handful of them involve hash functions, some of which are presented in the following.

`OP_HASH160 [0xa9]`: The input is hashed twice: first with SHA-256 and then with RIPEMD-160.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_HASH160.P\ \rightarrow\ \widetilde{v} :: sha256(ripemd160(v_n)),\ P}\ \text{OP} - \text{HASH160}$$

`OP_HASH256 [0xaa]`: The input is hashed two times with SHA-256.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_HASH256.P\ \rightarrow\ \widetilde{v} :: sha256(sha256(v_n)),\ P}\ \text{OP} - \text{HASH256}$$

Cryptographic hash-puzzles

With these few opcodes we can already create our own scriptPubKey, such that it becomes spendable by solving a hash-puzzle.
Lets create the following lock-script.

```
scriptPubKey:  OP_HASH256 <2fcdec9164a29c9986114244a
               21b81cc9f459249553a716034d00d1a6ce5b
               5f2> OP_EQUAL
```

Note that if the opcode `OP_HASH160` would have been used instead of `OP_HASH256`, then the scriptPubKey would have been interpreted as a Pay-To-Script-Hash as discussed previously! But this transaction output instead becomes spendable, if provided with just some data that hashed twice returns `2fcdec9164a29c9986114244a21b81cc9f459249553a7160 34d00d1a6ce5b5f2`. This is generally a very hard problem due to the sheer amount of possible inputs. But with the hint "The colour of the sky", we can come up with the right answer: blue! The string "blue" hashed twice returns the hash-value from above.

We can then create a transaction spending the output by providing the scriptSig `<blue>`. The evaluation of the complete Script is thus `<blue> OP_HASH256 <2fcdec9164a29c9 986114244a21b81cc9f459249553a716034d00d1a6ce5b5f2> OP_EQUAL` and can be seen in Table 6.2.1.

This leaves true on the value stack after a successful termination of the Script program, and the transaction would be valid.

Although this program may seem simplistic and of limited use, the principle behind it is needed to construct complex applications, which will be discussed in Chapter 8.

Checking signatures

One of the most important but also complex opcode is `OP_CHECKSIG`. We have already encountered it in one of the examples before. Its purpose is to make sure that the spending transaction must provide a valid signature from the recipient of the funds. It expects a signature and public key on the value stack when executed.

Table 6.2.1: The Script program of a transaction containing a hash puzzle

Stack	Script	Description
`(empty)`	`<blue> OP_HASH256 <2fcd...> OP_EQUAL`	The *scriptSig* and *scriptPubKey* is combined.
`<blue>`	`OP_HASH256 <2fcd...> OP_EQUAL`	The constant `<blue>` is pushed on the stack.
`<2fcd...>`	`<2fcd...> OP_EQUAL`	The top item `<blue>` is hashed twice.
`<2fcd...> <2fcd...>`	`OP_EQUAL`	The given hash is pushed to the stack.
`true`	`(empty)`	The hashes are compared, leaving true on the stack.

The formalization of `OP_CHECKSIG` involves a function $SignatureChk$, which returns true if the signature is valid, and which will be explained later.

`OP_CHECKSIG [0xac]`: Verifies signature and public key against the transaction spending it.

$$\frac{SignatureChk[t = (\widetilde{I}, \widetilde{O}, l)] = true}{\widetilde{v} :: \text{<sig>} :: \text{<pubkey>},\ OP\_CHECKSIG[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v} :: 1,\ P,}\ \text{OP} - \text{CHECKSIG1}$$

$$\frac{SignatureChk[t = (\widetilde{I}, \widetilde{O}, l)] \neq true}{\widetilde{v} :: \text{<sig>} :: \text{<pubkey>},\ OP\_CHECKSIG[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v} :: 0,\ P,}\ \text{OP} - \text{CHECKSIG0}$$

To validate the provided signature, `OP_CHECKSIG` needs information about the transaction from the context. This information is provided by the annotation. It contains the spending transaction itself, $t = (\widetilde{I}, \widetilde{O}, l)$.

There is an opcode called `OP_CHECKMULTISIG`, which instead of checking one signature against the transaction for a public key checks a specified number of signatures for different public keys. This is also called M-of-N multisignature.

`OP_CHECKMULTISIG [0xae]`: If all signatures are valid, 1 is returned, 0 otherwise. Due to a bug in the original implementation of Script, one extra unused value is removed from the stack. Therefore the stack must contain one extra value from below, which is usually just set to 0. In the execution rule below, this value is simply denoted as v_{drop}.

$$\frac{SignatureMultiChk[t = (\widetilde{I}, \widetilde{O}, l)] = true}{\begin{aligned} \widetilde{v} :: v_{drop} :: \; & < sig1 >:: \cdots ::< sigM >::< M >:: \\ & < pub1 >:: \cdots ::< pubN >::< N >, \\ & OP\_CHECKMULTISIG[t = (\widetilde{I}, \widetilde{O}, l)].P \\ & \rightarrow \; \widetilde{v} :: 1, \; P, \end{aligned}} \; \text{OP} - \text{CHECKMULTISIG1}$$

$$\frac{SignatureMultiChk[t = (\widetilde{I}, \widetilde{O}, l)] \neq true}{\begin{aligned} \widetilde{v} :: v_{drop} :: \; & < sig1 >:: \cdots ::< sigM >::< M >:: \\ & < pub1 >:: \cdots ::< pubN >::< N >, \\ & OP\_CHECKMULTISIG[t = (\widetilde{I}, \widetilde{O}, l)].P \\ & \rightarrow \; \widetilde{v} :: 0, \; P, \end{aligned}} \; \text{OP} - \text{CHECKMULTISIG0}$$

Locktime

The locktime of a transaction was mentioned earlier. It is the value of a transaction that denotes the earliest time after which it can be accepted. But it is possible to also use opcodes that programmatically enforce a spending transaction to contain a locktime of at least some specific value. The locktime is a 4 byte unsigned integer, and therefore can be between 0 and $4.294.967.295$. As discussed in Section 4.2, when the value is below $500.000.000$ it represents the earliest block number after which the transaction can be confirmed in the blockchain. Otherwise, it denotes a UNIX timestamp and the latest date that can be encoded as a timestamp in the locktime is the 7th February, 2106.

To check the locktime against a value specified in the scriptPubKey, the opcode `OP_CLTV [0xb1]` is used. It needs to receive this transaction's locktime through an annotation. The function that retrieves the locktime from the annotation is called $LocktimeChk$.

Just as for the opcode `OP_SIZE`, the opcode `OP_CLTV` does not remove its operand. The reason for this behaviour is due to some historic background. This opcode was introduced with a soft fork to repurpose the skip operation `OP_NOP2`. Old clients will therefore still regard `OP_CLTV` as a skip operation, while new clients will understand its new purpose and make sure that it evaluates to true. Hence both old and new clients can still continue to be part of the same network, while new clients are more restrictive and as a majority can enforce the new restrictions. It would be good to have an opcode that instead simply pushes its result onto the stack. But introducing `OP_CLTV` with a soft fork did limit the possibilities of the opcode to simply either evaluate to true and continue execution, or immediately mark the transaction as invalid.

$$\frac{LocktimeChk[t = (\widetilde{I}, \widetilde{O}, l)] = true}{\widetilde{v} :: \text{<locktime>},\ OP\_CLTV[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v},\ P,}\ \text{OP} - \text{CLTV1}$$

$$\frac{LocktimeChk[t = (\widetilde{I}, \widetilde{O}, l)] = false}{\widetilde{v} :: \text{<locktime>},\ OP\_CLTV[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v},\ \text{invalid tx},}\ \text{OP} - \text{CLTV0}$$

Just as for the locktime, a similar opcode exists for the sequence number. The function that can retrieve the sequence number through the annotation is called $SequenceChk$. Recall, that whereas the locktime is a value specified for the whole transaction, the sequence number is specified for each input and can differ. The opcode is called `OP_CSV [0xb2]`.

$$\frac{SequenceChk[t = (\widetilde{I}, \widetilde{O}, l)] = true}{\widetilde{v} :: \text{<sequence>},\ OP\_CSV[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v},\ P,}\ \text{OP} - \text{CSV1}$$

$$\frac{SequenceChk[t = (\widetilde{I}, \widetilde{O}, l)] = false}{\widetilde{v} :: \text{<sequence>},\ OP\_CSV[t = (\widetilde{I}, \widetilde{O}, l)].P,\ \rightarrow\ \widetilde{v},\ \text{invalid tx},}\ \text{OP} - \text{CSV0}$$

The functions $LocktimeChk$ and $SequenceChk$ check the actual locktime, or sequence number, of the transaction t against the value on the stack. But there are a few more rules, which are described in the proposal of the corresponding opcodes in BIP65 [22] and BIP112 [6]. All the needed values are available through the stack or the annotation containing the whole transaction t. We will therefore just informally describe the $LocktimeChk$ ad $SequenceChk$ functions.

The function $LocktimeChk$ will return `false` if

- the stack is empty, or
- the top item on the stack is less than 0, or
- the locktime type (height vs. timestamp) of the top stack item and the locktime are not

the same, or

- the top stack item is greater than the transaction's locktime field, or
- the sequence field of the transaction input is 0 (disables the locktime for this input).

Otherwise $LocktimeChk$ returns `true`.
Likewise for $SequenceChk$, rules dictate that the function returns `false` if

- the stack is empty, or
- the top item on the stack is less than 0, or
- the top item on the stack has the disable flag (1 ≪ 31) unset, and
  - the transaction version is less than 2, or
  - the transaction input sequence number disable flag (1 ≪ 31) is set, or
  - the relative locktime type is not the same, or
  - the top stack item is greater than the transaction sequence.

Otherwise $SequenceChk$ returns `true`. Recall from Chapter 4 that the current interpretation used for the sequence number remains impractical. However, the rule and opcode described above could become very meaningful when implemented, and are expected to be part of the consensus rules of Bitcoin within the next few months. We therefore find it appropriate to design the model with this functionality in mind.

The sequence number denotes a relative time that must have passed before the output can be spent. This is also refereed to as the reach of "maturity" for the parent transaction output. But the sequence number is not simply a 4 byte unsigned integer like the locktime. Whereas for the locktime, the distinction between a block representation and a timestamp was solely based on its value, the sequence number is instead repurposed by BIP68 [14], such that the distinction between a relative block representation and relative timespan is determined by the 22nd bit of the sequence number. When this value is set to 1, the sequence number denotes a timespan in units of 512 seconds granularity, which is chosen since a block is mined roughly every 600 seconds. The specification only interprets 16 bits of the sequence number as relative lock-time, and allows for a year of relative lock-time, while the remaining bits of the sequence number allow for future expansion of the sequence number.

Flow control

All the opcodes presented did so far not use the Boolean execution stack $\widetilde{B}$. The following opcodes regard the flow control and explain how they manipulate and use this special stack.

The flow control has until now just been ignored by assuming that all opcodes will be executed. But the opcodes must only execute if it is not present in an if/else-clause, or if the clause statement was true. We will add a special rule called `BOOL-RELATION` to each of the opcodes.

To check if the execution is currently in an if/else-clause, an opcode $o \notin$ { `OP_IF`, `OP_NOTIF`, `OP_ELSE`, `OP_ENDIF` } must check the top element of $\widetilde{B}$, and only execute if it differs from zero. Otherwise, the opcode will be skipped and execution of the program continues with the next opcode, using of course the same rule.

`BOOL-RELATION`: General flow control rule. If not currently in an if/else-clause, or prevailing clause-statement was true, then execute opcode:

$$\frac{o \notin \{\,\text{OP\_IF},\ \text{OP\_NOTIF},\ \text{OP\_ELSE},\ \text{OP\_ENDIF}\,\} \quad top(\widetilde{B}) \neq 0 \quad \widetilde{v},\ o.P \ \rightarrow\ \widetilde{v}',\ P}{\widetilde{v},\ o.P,\ \widetilde{B}\ \rightarrow\ \widetilde{v}',\ P,\ \widetilde{B}}$$

`BOOL-RELATION-SKIP`: General flow control rule. If the statement for the if/else-clause was not true, then skip opcodes within this clause:

$$\frac{o \notin \{\,\text{OP\_IF},\ \text{OP\_NOTIF},\ \text{OP\_ELSE},\ \text{OP\_ENDIF}\,\} \quad top(\widetilde{B}) = 0 \quad \widetilde{v},\ o.P \ \rightarrow\ \widetilde{v},\ P}{\widetilde{v},\ o.P,\ \widetilde{B}\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B}}$$

To indicate the execution of an if/else-clause, the opcodes for the flow control `OP_IF`, `OP_NOTIF`, `OP_ELSE` and `OP_ENDIF` must manipulate the Boolean stack $\widetilde{B}$ accordingly.

But just as for the other opcodes above, also all the opcodes for the flow control execute only if the top element of $\widetilde{B}$ is not zero, meaning that they are not being executed in an if/else-clause, or that this part of the clause must execute. The opcode `OP_IF` must decide if the opcodes within its clause must execute or not. It is decided by checking if the top element of the value stack is not zero, meaning that the if-statement is true. To indicate that this part must execute, 1 is then pushed onto the Boolean stack $\widetilde{B}$ and the top value of the value stack $\widetilde{v}$ is removed.

`OP_IF [0x63]`:

$$\frac{v_n \neq 0 \quad top(\widetilde{B}) \neq 0}{\widetilde{v} :: v_n,\ OP\_IF.P,\ \widetilde{B}\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B} :: 1}\ \mathrm{OP-IF1}$$

If the top element of the value stack is 0, meaning that the guard is false, then 0 is pushed onto $\widetilde{B}$ to indicate that this if/then-clause must not be executed.

$$\frac{top(\widetilde{B}) \neq 0}{\widetilde{v} :: 0,\ OP\_IF.P,\ \widetilde{B}\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B} :: 0}\ \mathrm{OP-IF0}$$

If the value stack is empty, then the transaction is marked as invalid.

$$\frac{}{\emptyset,\ OP\_IF.P,\ \widetilde{B}\ \rightarrow\ \emptyset,\ \text{invalid tx},\ \emptyset}\ \mathrm{OP-IFINVALID}$$

There also exists an opcode `OP_NOTIF`, which is just the negated version of `OP_IF`. Its technical specification is similar to that of `OP_IF` and can be found in the appendix.

Script allows for consecutive `OP_ELSE [0x67]` to appear and execute. If the preceding `OP_IF`, `OP_NOTIF` or `OP_ELSE` was not executed, then the statements following an `OP_ELSE` are, and if the preceding `OP_IF`, `OP_NOTIF` or `OP_ELSE` was executed then the statements are not. Just as for `OP_IF`, a 1 or 0 is pushed onto the Boolean execution stack $\widetilde{B}$ to indicate that the following opcodes must be executed, or not.

$$\frac{}{\widetilde{v},\ OP\_ELSE.P,\ \widetilde{B} :: 0\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B} :: 1}\ \mathrm{OP-ELSE1}$$

$$\frac{}{\widetilde{v},\ OP\_ELSE.P,\ \widetilde{B} :: 1\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B} :: 0}\ \mathrm{OP-ELSE0}$$

If `OP_ELSE` is called without a preceding `OP_IF` or `OP_NOTIF`, then $\widetilde{B}$ must have been empty. The whole transaction is then immediately marked as invalid.

$$\frac{}{\widetilde{v},\ OP\_ELSE.P,\ \emptyset\ \rightarrow\ \widetilde{v},\ \text{invalid tx},\ \emptyset}\ \mathrm{OP-ELSEINVALID}$$

The end of an if/else-clause is marked by the opcode `OP_ENDIF [0x68]`. It ends an if/else block by popping the top item of $\widetilde{B}$. All if/else-clauses must end, otherwise the transaction is marked invalid. Having an `OP_ENDIF` without a preceding `OP_IF` will be marked invalid by the fact that $\widetilde{B}$ must be empty. At the latest when a Script program ends, any unbalanced if/else-clause will be recognized by the Boolean execution stack $\widetilde{B}$ being non-empty.

$$\frac{}{\widetilde{v},\ OP\_ENDIF.P,\ \widetilde{B} :: b_n\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B}}\ \text{OP} - \text{ENDIF}$$

$$\frac{}{\widetilde{v},\ OP\_ENDIF.P,\ \emptyset\ \rightarrow\ \widetilde{v},\ \text{invalid tx},\ \emptyset}\ \text{OP} - \text{ENDIFINVALID}$$

Furthermore, there are two special opcodes. One is called `OP_RETURN [0x6a]` and it will immediately mark the transaction as invalid, but can be used to store small data chunks of up to 80 bytes. A simple application for this has in fact been used to timestamp this thesis, see the colophon on the verso of the title page. By using `OP_RETURN`, its SHA256 hash digest has been added to the Bitcoin blockchain, providing proof that the thesis existed from the time that it was added. This whole process is done without the need of a central authority!

The other is `OP_VERIFY [0x69]` and checks if the top stack item is 0, and if so terminates the program marking the transaction invalid. All the following opcodes `OP_EQUAL`, `OP_NUMEQUAL`, `OP_CHECKSIG` and `OP_CHECKMULTISIG` exist as a variant called `OP_EQUALVERIFY[0x88]`, `OP_NUMEQUALVERIFY[0x9d]`, `OP_CHECKSIGVERIFY[0xad]` and `OP_CHECKMULTISIGVERIFY[0xaf]`, which do the same as their namesake suggest, but additionally execute the `OP_VERIFY` procedure afterwards. This is useful in programs that e.g. contain more than one check. Otherwise, only the last check in a program would be evaluated by the rule that marks the transaction valid only if the top stack value is `true`.

$$\frac{}{\widetilde{v},\ OP\_RETURN.P,\ \widetilde{B}\ \rightarrow\ \emptyset,\ \text{invalid tx},\ \emptyset}\ \text{OP} - \text{RETURN}$$

$$\frac{}{\widetilde{v} :: 1,\ OP\_VERIFY.P,\ \widetilde{B}\ \rightarrow\ \widetilde{v},\ P,\ \widetilde{B}}\ \text{OP} - \text{VERIFY1}$$

$$\frac{}{\widetilde{v} :: 0,\ OP\_VERIFY.P,\ \widetilde{B}\ \rightarrow\ \emptyset,\ \text{invalid tx},\ \emptyset}\ \text{OP} - \text{VERIFY0}$$

Special case

Recall, that there exist a "special" transaction output called Pay-To-Script-Hash (P2SH), see Section 6.1. The scriptPubKey will be executed by the execution rules defined above and must evaluate to true. The scriptPubKey can be recognized as a P2SH by its pattern, namely if the scriptPubKey is exactly `OP_HASH160 <20-byte-hash> OP_EQUAL`. The value `20-byte-hash` is of course pushed by the opcode [0x14], but we will just use the notation specified by "<>"

as usual.

An additional execution procedure will be performed for a P2SH. Informally, from the initial scriptSig the last push opcode is removed, which otherwise pushes the serialized redeemScript to the stack. The scriptSig is then executed simply with that push opcode removed, and must evaluate to true, in which case spending of the P2SH output is accepted.

6.3 Signature checking

The part of `OP_CHECKSIG` that makes it complex is the $SignatureChk$ function. To understand it, we need to take a look at how transactions can be represented as strings and then be signed. Every transaction can be represented as a long hex string, referred to as the *serialization* of the transaction. All the variable fields of the transaction are simply listed one after the other, and they correspond directly to the values of the data structure for the transaction. This is possible, as for each variable that contains information there is a preceding variable that specifies its length. The signature can then be produced for this serialization of the transaction.

Walk-through example of a serialized transaction

This section will give a very low-level example of how serialized transactions are composed. It follows the exact same transaction structure already defined in Section 4.2, and which can also be found in Table 4.2.1.

Every serialization of a transaction begins with a 4-byte version field that can specify the rules this transaction follows. Currently the version is just 1.

Then all data fields for the transaction inputs are appended. The first 1-byte denotes the number of inputs this transaction has. Then the 32-byte hash of a transaction which we want to spend an output from is specified. This is followed by a 4-byte field that denotes the output index we want to spend from. Recall that the output index of a transaction starts from zero.

Then the scriptSig data for this input follows, which starts with a 1-byte field denoting the length of the script itself. The actual scriptSig is added in its hex string representation, which must of course be of exactly the same size as specified with the 1-byte field before. Then a 4 byte field denoting the sequence number is specified for the input, which is typically set 0 to deactivate its function as a relative locktime.

If the transaction has more than one input, as denoted with the field containing the number of inputs, then the data for the next input is specified in the same way: starting again with the 32-byte hash of the transaction, which this input references, followed by the output index, scriptSig size, and actual scriptSig data.

After all the inputs are specified, the outputs will be defined. The next 1-byte field contains the number of outputs in the transaction.

The following 8-bytes contain a field that determines the amount of bitcoins being send with this first output, represented as a 64-bit integer. For example 1 BTC, or 100.000.000 Satoshis, is represented as `0x5f5e100`. Then the scriptPubKey data for that output is provided. Again a 1-byte field first denotes the length of the scriptPubKey. Then the actual scriptPubKey is added in its hex string representation, which must of course be of exactly the same size as specified with the 1-byte field before. Again this process is repeated for all the outputs, starting from the field containing the amount, then the scriptPubKey size and scriptPubKey data.

Finally the 4-byte locktime of the transaction is added, which if set to zero means that the transaction may be added to a block immediately.

The string containing all of this information corresponds to the serialization of a transaction. Table 6.3.1 shows some real data as an example. The transaction from the example contains only one input, and therefore spends exactly one output. The output it spends is a Pay-To-PubKey-Hash. The scriptSig must therefore provide a signature and public key.

The scriptSig of the input to the transaction is shown in Table 6.3.1, and the signature can of course be obtained directly from it. The scriptSig starts with the opcode in hex `[0x49]`, which is an unlabeled push opcode, pushing the next $0x49 = 73$ bytes of data onto the stack. Those next 73 bytes of data in the scriptSig form the signature, namely `30460221009e0339f72c793a89e664a8a932df073962a3f84eda0bd9e02084a6a9567f75aa022100bd9cbaca2e5ec195751efdfac164b76250b1e21302e51ca86dd7ebd7020cdc0601`.

It is followed by an opcode with the hex `[0x41]`. This opcode pushes the next $0x41 = 65$ bytes of data onto the stack, namely `0450863ad64a87ae8a2fe83c1af1a8403cb53f53e486d8511dad8a04887e5b23522cd470243453a299fa9e77237716103abc11a1df38855ed6f2ee187e9c582ba6`. Those 65 bytes of data are the public key to the Bitcoin address that received the funds that are being spend with this transaction. The scriptSig is therefore of the typical form `<sig> <pubkey>`, and satisfies the condition of the Pay-To-PubKey-Hash output. But the question that remains is: how can a signature for a serialized transaction be produced or verified?

Table 6.3.1: A serialized transaction containing one input and one output

Version:	`01000000`
Total input number:	`01`
Transaction hash:	`eccf7e3034189b851985d871f91384b8ee35`
	`7cd47c3024736e5676eb2debb3f2`
Output index[a]:	`01000000`
ScriptSig size:	`8c`
ScriptSig:	`4930460221009e0339f72c793a89e664a8a9`
	`32df073962a3f84eda0bd9e02084a6a9567f`
	`75aa022100bd9cbaca2e5ec195751efdfac1`
	`64b76250b1e21302e51ca86dd7ebd7020cdc`
	`0601410450863ad64a87ae8a2fe83c1af1a8`
	`403cb53f53e486d8511dad8a04887e5b2352`
	`2cd470243453a299fa9e77237716103abc11`
	`a1df38855ed6f2ee187e9c582ba6`
Sequence number[b]:	`ffffffff`
Total output number:	`01`
Amount[c]:	`00e1f50500000000`
ScriptPubKey size:	`19`
ScriptPubKey:	`76a914097072524438d003d23a2f23edb65a`
	`ae1bb3e46988ac`
Locktime:	`00000000`

All data appear as hexadecimals.

[a] The output index starts at zero. Therefore the first output of a transaction has always index 0.

[b] The sequence number is currently disabled in the Bitcoin protocol, and is therefore typically set to 0 =`0xffffffff`. But as discussed previously, in future it may be repurposed.

[c] The amount must be a 64-bit integer, denoted in Satoshis.

Generation and verification of transaction signatures

Production and verification of signatures follow steps that are similar to each other. For producing the signature, any serialized transaction can be created as shown above, but the fields of the scriptSig size and data in all the inputs are just left empty. When verifying a signature, the transaction is received in its serialized form as above, and the fields for scriptSig and scriptSig size must be set to empty.

Now follows an additional step, which is cumbersome and simply has to be accepted, since the signature process in Bitcoin is defined in this way. For the input under consideration, we insert into the empty fields of the scriptSig size and data the scriptPubKey of the output, which is being spend with this input. It is important to note that this scriptPubKey does *not* come from the transaction we are checking the signature for. It is the scriptPubKey originating from the transaction output referenced with the input that we are checking the signature of, and hence the one that the scriptSig provides the (valid) data for. To keep the confusion as small as possible, we will in the following denote this scriptPubKey as *SubScript*.

From this modified serialized transaction the signature can be verified as explained using the public key, or produced with the private key.

Specification of the *SignatureChk* function

The $SignatureChk$ function used by the rule $\mathrm{OP}-\mathrm{CHECKSIG}$ in our model implements the ECDSA signature algorithm to check signatures against public keys. The function can now be formalized as follows.

Function $signatureChk[t = (\widetilde{I}, \widetilde{O}, l)]$:

1. Pop the top two elements from stack $\widetilde{v}$, which are expected to be `<pubkey>` and `<sig>`.

2. Create $t' = (\widetilde{\emptyset}_i, \widetilde{O}, l)$, where
   $\widetilde{\emptyset}_i = (\emptyset_1, \ldots, \emptyset_{i-1},\ I'_i,\ \emptyset_{i+1}, \ldots, \emptyset_n)$
   $\emptyset_j = (thash_j, n_j, \emptyset)$, So $\emptyset_j$ is input I_j, but with the scriptSig removed.
   $I'_i = (thash_i, n_i, SubScript)$ So I'_i is input I_i, but with its scriptSig replaced by SubScript.

3. Hash the serialized data of transaction t' using the SHA256 function twice, yielding h'.

4. Return `true` if the signature `<sig>` can be accepted by the ECDSA signature verification algorithm for h' and `<pubkey>`, otherwise `false`.

Signature types

The process above describe in principle exactly how signatures are checked and produces in Bitcoin. But there is one detail missing. The signature additionally contains a 1-byte signature type appended to it. The signature type specifies how the signature was produced and must be verified. The procedure above is only used when this type is of a value representing the type called `SIGHASH_ALL`.

The actual data containing the signature is therefore only the part until the last byte of the pushed data denoted as `<sig>` and used in the $SignatureChk$ function for the signature. This push containing the signature data can therefore instead simply be expressed as `<sig|hashType>`, where the symbol | denotes the concatenation of the signature `sig` with the 1-byte signature type `hashType`.

Depending on the value of the signature type, the serialized transaction used to produce or verify a signature must be modified.

There exist three different signature types, and one additional option that can be set for each of them.

- `SIGHASH_ALL`,
- `SIGHASH_NONE`,
- `SIGHASH_SINGLE`, and
- an additional option called `SIGHASH_ANYONECANPAY`.

To address all cases, the $SignatureChk$ function must be modified accordingly. We will first look at the procedures for the first three types. The last type called `SIGHASH_ANYONECANPAY` is an optional "setting", which must be combined with any one of the first three types. Steps 1, 2 and 4 of the function stay the same. But step 3 changes as follows:

- If `hashType = 0x01` (`SIGHASH_ALL`), then we are in the case already described above and can proceed as described. This can be thought of as the process that "*signs all the outputs*"

- If `hashType = 0x02` (`SIGHASH_NONE`), then all the outputs of the serialized transaction are replaced with empty vectors. This can be thought of as "*sign none of the outputs – I don't care where the bitcoins go.*"
  $t' = (\widetilde{\emptyset}_i, \widetilde{O})$, where
  $\widetilde{\emptyset}_i$ is as defined for `SIGHASH_ALL` above.
  $$\begin{aligned} \widetilde{O} &= (\overbrace{\underline{\emptyset}, \dots, \underline{\emptyset}}^{\text{m times}}) \\ \underline{\emptyset} &= (\emptyset, \emptyset) \end{aligned}$$

- If `hashType = 0x03` (`SIGHASH_SINGLE`), then the transaction must have the same number of inputs as outputs. All outputs with indices different from the input are replaced with empty vectors. This can be thought of as "*sign one of the outputs – I don't care where the other outputs go*".
  $t' = (\widetilde{\emptyset}_i, \widetilde{O})$, where
  $\widetilde{\emptyset}_i$ is as defined for `SIGHASH_ALL` above.
  $$\begin{aligned} \widetilde{O} &= (\overbrace{\underline{\emptyset}, \dots, \underline{\emptyset}}^{\text{i-1 times}}, O_i) \\ \underline{\emptyset} &= (\emptyset, \emptyset) \end{aligned}$$

- If additionally the `SIGHASH_ANYONECANPAY` option is set, then `hashType` must be either `0x81`, `0x82`, or `0x83`. In other words, if the first "half-byte" of any of the previous three signature types is `8` instead of `0`, then the additional process for `SIGHASH_ANYONECANPAY` is used. The output $\widetilde{O}$ used in t' is defined by one of the three signature types from above, but the input vector is resized to one.
  $t' = (I'_i, \widetilde{O})$, where
  I'_i is as defined for `SIGHASH_ALL` above.
  $\widetilde{O}$ is as defined by the signature types above.
  Note that t' in `SIGHASH_ANYONECANPAY` is produced solely to contain the input I'_i, which is under consideration. Therefore adding this optional setting to one of the three types from above can be thought of as "*Let other people add inputs to this transaction – I don't care where the rest of the bitcoins come from*".

The signature type is then appended to the serialized transaction t' and double hashed using SHA256. This resulting hash value h' is then used to check against the signature against and public key as described above, or to generate the signature in the first place.

Most use cases only comprise the `SIGHASH_ALL` type. But the different signature types make the signature checking flexible, because the transaction that is signed can be controlled through the use of the types. In this way, e.g. contracts can be constructed in which each party only signs some part of it, allowing other parts to be changed without their involvement.

Specification of the *SignatureMultiChk* function

The $SignatureMultiChk$ used for the rules of $\mathrm{OP-CHECKMULTISIG}$ basically reiterates over the function $SignatureChk$. It takes the N public keys and the M signatures, and starting from the first signature it compares it using the $SignatureChk$ function to the first public key. If it returns true, then it will check the next signature against the next public key. If instead it returns false, then it will continue checking the signature against the next public key, until eventually finding a match. If using this approach does not yield a match for any one of the signatures, then $SignatureMultiChk$ returns false. Otherwise it returns true.

Note that this approach requires the provided signatures to be ordered in regard to the public keys. Once a public key was checked against a signature, it will not be checked against any other signature again. This is in spite of a check of a public key being successful or not. It is a simple procedure, which was probably implemented this way in Bitcoin to avoid the check of signatures to grow exponentially with the number of public keys involved.

I am very intrigued by Bitcoin. It has all the signs. Paradigm shift, hackers love it, yet it's derided as a toy. Just like microcomputers.

Paul Graham

7 Variables and abstractions

As discussed previously, Script is a low-level stack-based Turing incomplete programming language, which contains basic operations but lacks abstractions and functions such as declarations of variables. Most programming languages come with compilers that map human friendly code to machine interpretable operations, also called *opcodes* that we met in Chapter 6.

In this chapter a high-level language for Script will be developed called NextScript, which can be compiled to Script. It will have an easier readability and use of expressions through semantics and abstractions. Numbers will for example always be represented in smallest bytes and pushed using the smallest possible opcodes. This is favourable, since it gives compact Script programs and also follows the current progress in standardizing transactions described in the Bitcoin protocol, by e.g. the *BIP62* proposal [25]. Furthermore, through a type system NextScript can catch obvious typing errors and guarantees the use of correct data types in programs. Hence, a sum will for example only be calculated on two numerical values. Script programs that would otherwise calculate the sum of hash values are prevented from being programmed with NextScript. We will begin this chapter by introducing variables and expressions, which is an improvement to the elementary stack that Script uses, and a simplification when dealing with data.

We will then look at commands and define the syntax of a NextScript program with its grammar. Finally, a type system is introduced. Suggestions for optimizations on the compiler are presented at the end of this chapter.

7.1 Variables

Script does not offer the functionality of declaring or using variables. It is therefore very difficult to manually keep track of what values are at which positions on the stack during any moment of the runtime of a program. Although Script programs are typically compact and somewhat limited in size, a transaction output can become permanently unspendable by accidentally popping one wrong value from the stack.

Since all values accessible to a Script program are on the value stack, variables can be introduced by keeping track of where on the stack the data for a variable can be found. A function $f : X \rightharpoonup \mathbb{N}$ can be defined to return the position in the stack where the data for a variable $x \in X$ is currently stored. This function will be called a *variable function* and its assignments of variables to positions will be called *pointers*. Furthermore, a variable function is said to be *1-free* if there is no variable with a pointer to 1 in the function. If a variable function is 2-free, it means that no variables are assigned pointers to 1 and 2, and so forth.

The compiler of NextScript will take two arguments, a NextScript program P and a variable function f. It returns a Script program S, denoted $S = \llbracket P \rrbracket_f$.

Compiling variables

We will now take a look at how data for a variable x can be computed and will return to declaring variables later. For optimization reasons, the compilation involves three cases depending on where in the stack the data is stored. In each case, the data of the variable is copied to the top of the stack. After the compilation, all the pointers in the variable function will therefore have increased by one, denoted as f_{+1}. In other words, $f_{+1}(x) = f(x) + 1$. A variable function f_{+1} is therefore at least 1-free, which follows directly from the definition.

$$\begin{array}{lll} [\![x]\!]_f = & \texttt{OP\_DUP} & \text{if } f(x) = 1 \\ [\![x]\!]_f = & \texttt{OP\_OVER} & \text{if } f(x) = 2 \\ [\![x]\!]_f = & \texttt{<}f(x)\texttt{> OP\_PICK} & \text{if } f(x) \geq 3 \end{array}$$

We could have used the opcode OP_PICK in all the cases. But OP_PICK requires an additional byte in the form of a push opcode, which defines from where in the stack the value should be obtained. To obtain smaller Script programs, it is therefore better to make it case dependent, and use the opcodes that require the fewest bytes.

In the following we will see that from the syntax of NextScript, whenever a variable is called, the copy of its value is immediately consumed by some operation in Script. But although a variable is not used anymore, its data will remain somewhere in the stack. This does not matter. When a Script program ends, only the top stack item is checked in a final step to decide if the transaction is valid or not. This compares favourably to popping the unused values, which would otherwise bloat the program size.

7.2 Expressions

Expressions contain strings, hash functions that manipulate strings, numeric values, arithmetic functions, and logic operations to e.g. compare expressions.

Hash functions and strings

String are just defined by encumbering them in double quotes (“”). Hash values are expressed as string values, and therefore hash functions return strings. Note that some applications, such as e.g. hash tables, need to manipulate hash values arithmetically. But numeric values are restricted to 4 bytes in Script, while hash values are of at least 20 bytes. In Script therefore no operation between hashes and numeric values can be done in any meaningful way. The restriction of such operations is introduced with the type system in Section 7.4.

$$\begin{array}{ll} h := & ripemd(h) \quad | \quad sha1(h) \quad | \quad sha256(h) \quad | \quad hash160(h) \quad | \\ & hash256(h) \quad | \quad "string" \end{array}$$

As already seen in Section 6.2, data in Script is pushed using opcodes OP_PUSHDATA1, OP_PUSHDATA2 and unlabeled opcodes [0x01] to [0x4b]. The hexadecimal value of the unlabeled opcodes represent the number of following bytes to push, see Section 6.2 for more details. The opcodes `OP_PUSHDATA1` and `OP_PUSHDATA2` must be used to push data greater than $0x4b = 75$ bytes. As previously discussed, the maximum size of a data push is limited to 520 bytes.

Depending on the size of the specific string to be pushed, the compilation is as follows.

$[\!["string"]\!]_f$	=	`[0x01-0x4b]` $string$	If $string$ size < 75 bytes
$[\!["string"]\!]_f$	=	`OP_PUSHDATA1` `[0x4c-0xff]` $string$	If $75 \leq string$ size < 255 bytes
$[\!["string"]\!]_f$	=	`OP_PUSHDATA2` `[0x0100-0x0208]` $string$	If $255 \leq string$ size ≤ 520 bytes

In each case, exactly one element is pushed onto the stack.

The available hash functions are:

- $ripemd$: Uses the RIPEMD-160 hash function
- $sha1$: Uses the SHA-1 hash function
- $sha256$: Uses the SHA-256 hash function
- $hash160$: Uses first the SHA-256 hash function and then the RIPEMD-160 hash function on that result
- $hash256$: Uses the SHA-256 hash function twice

The hash functions are simply compiled using build-in opcodes of Script.

$[\![ripemd160(h)]\!]_f$	$=$	$[\![h]\!]_f$ `OP_RIPEMD160`
$[\![sha1(h)]\!]_f$	$=$	$[\![h]\!]_f$ `OP_SHA1`
$[\![sha256(h)]\!]_f$	$=$	$[\![h]\!]_f$ `OP_SHA256`
$[\![hash160(h)]\!]_f$	$=$	$[\![h]\!]_f$ `OP_HASH160`
$[\![hash256(h)]\!]_f$	$=$	$[\![h]\!]_f$ `OP_HASH256`

After the compilation of any expression, exactly one element is pushed onto the stack. This is easily seen from the few possible cases of the expressions above. First the expression h is computed, which at some point must result in a single push. An opcode for a hash function pop that element and push its hash value onto the stack, leaving the remainder of the stack unchanged. This will later be proven formally as a Lemma for all expressions.

Expressions

We will now add arithmetic values and operations to the expressions, which are all based on integers.

$$\begin{aligned} e, e_1, e_2 := \quad & h \quad | \quad \|h\| \quad | \quad integer \quad | \\ & e_1 + e_2 \quad | \quad e_1 - e_2 \quad | \quad -e \quad | \quad |e| \quad | \quad e{+}{+} \quad | \quad e{-}{-} \quad | \\ & min(e_1, e_2) \quad | \quad max(e_1, e_2) \end{aligned}$$

In the expression above, x denotes a variable, and h a string expression from before. The expression $\|h\|$ returns the string length of the string value evaluated from h. Here we need to be more careful. The Script opcode OP_SIZE returns the length of a value but without consuming it. To have a more streamlined proceeding, the compilation of this expression will instead consume the value it pushes the length of.

$[\![compute(\|h\|)]\!]_f = [\![compute(h)]\!]_f$ OP_SIZE OP_NIP

The opcode OP_SIZE first pushes the string length. Afterwards OP_NIP removes the second-to-top item from the stack, which is the value of the element whose length was just pushed.

The following expressions are integer values that are simply pushed onto the stack. For optimization reasons, the actual opcode used to push an integer must again depend on its value. After compiling any of them, the variable function will have increased its pointers by one.

$[\![integer]\!]_f$	$=$	OP_1NEGATE	if $integer = -1$
$[\![integer]\!]_f$	$=$	OP_0	if $integer = 0$
$[\![integer]\!]_f$	$=$	OP_1	if $integer = 1$
...			
$[\![integer]\!]_f$	$=$	OP_16	if $integer = 16$
$[\![integer]\!]_f$	$=$	[0x01-0x4b] $integer$	Otherwise

To push integers onto the stack, which are greater than 16 or smaller than -1, the unlabeled opcodes [0x01] to [0x4b] must be used. Those opcodes were also used to push strings. The value of $integer$ on the right hand-side is byte data and represented as a 64-bit signed integer. Since arithmetic functions are restricted to work on 4-byte integers, the integer is restricted to the range from $-2^{31} + 1$ to $2^{31} - 1$, and does therefore not use the whole theoretical range of the signed 64-bit integer representation. Although Script technically allows to push numeric values that are outside of this range, since no arithmetic operations can be done with those values in Script, it is not allowed in NextScript.

The number of $integer$ on the left-hand side will always be compiled to its smallest signed 64-bit representation. To push it using the smallest possible opcode, it is therefore case dependent on the integer. For example, the number 1 is pushed onto the stack using `OP_1`, but could technically also be pushed with `[0x01] 01`. They both do the same, but while `OP_1` is only one byte in size, `[0x01] 01` uses one byte for the opcode and one more for the number representation `01`, and is therefore in total two bytes.

The rest of the expressions are compiled as follows.

$$
\begin{array}{lcl}
[\![e_1 + e_2]\!]_f & = & [\![e_1]\!]_f \, [\![e_2]\!]_{f_{+1}} \ \texttt{OP\_ADD} \\
[\![e_1 - e_2]\!]_f & = & [\![e_1]\!]_f \, [\![e_2]\!]_{f_{+1}} \ \texttt{OP\_SUB} \\
[\![|e|]\!]_f & = & [\![e]\!]_f \ \texttt{OP\_ABS} \\
[\![e++]\!]_f & = & [\![e]\!]_f \ \texttt{OP\_1ADD} \\
[\![e--]\!]_f & = & [\![e]\!]_f \ \texttt{OP\_1SUB} \\
[\![min(e_1, e_2)]\!]_f & = & [\![e_1]\!]_f \, [\![e_2]\!]_{f_{+1}} \ \texttt{OP\_MIN} \\
[\![max(e_1, e_2)]\!]_f & = & [\![e_1]\!]_f \, [\![e_2]\!]_{f_{+1}} \ \texttt{OP\_MAX}
\end{array}
$$

As compiling e_1 adds one value to the stack, the compilation of e_2 uses the variable function f_{+1}. All the opcodes consume their operands and only push the result onto the stack. So, e.g. `OP_ADD` consumes two operands and pushes only the sum onto the stack.

The compilations above again satisfy the property that the variable function is increased by exactly one after each compilation of an expression. The proof of this is by induction on the size of the derivation.

Proposition. *Compiling an expressions adds exactly one element to the stack. Hence the variable function f is increased after the compilation of an expression. In other words, the variable function is f_{+1} after $[\![e]\!]_f$ is computed.*

Proof. The claim may be proven by an induction on the number of steps in the derivation of e and by showing that when compiled, the pointers in the variable function are increased by one.
Base case:
Derivation with 1 step. Then the expression consists of solely one terminal, namely a push of "$string$" or $integer$. From the definition of compiling to a push opcode, the pointers in the variable function are increased by exactly one and therefore the claim holds true for those cases.
Inductive hypothesis:

Suppose the claim holds true for all derivations with $n \geq 1$ steps.
Inductive step:
Prove that every derivation of e with $n+1$ steps when compiled increases the pointers by one for the variable function f.
The derivation may begin with

$$\begin{aligned}
\llbracket\, |e| \,\rrbracket_f &= \llbracket\, e \,\rrbracket_f \ \texttt{OP\_ABS}, \\
\llbracket\, e++ \,\rrbracket_f &= \llbracket\, e \,\rrbracket_f \ \texttt{OP\_1ADD}, \text{ or} \\
\llbracket\, e-- \,\rrbracket_f &= \llbracket\, e \,\rrbracket_f \ \texttt{OP\_SUB}.
\end{aligned}$$

Since the computations on the right hand-side can only be derived with n steps, by the inductive assumption, the pointers of the variable function f is increased by one. Since the opcodes `OP_ABS`, `OP_1ADD` and `OP_SUB` consume the top value and push the result of performing an operation on it, they do not change the variable function and therefore the claim holds true for these cases.

If instead the derivation starts with

$$\begin{aligned}
\llbracket\, e_1 + e_2 \,\rrbracket_f &= \llbracket\, e_1 \,\rrbracket_f \llbracket\, e_2 \,\rrbracket_{f+1} \ \texttt{OP\_ADD}, \\
\llbracket\, e_1 - e_2 \,\rrbracket_f &= \llbracket\, e_1 \,\rrbracket_f \llbracket\, e_2 \,\rrbracket_{f+1} \ \texttt{OP\_SUB}, \\
\llbracket\, min(e_1, e_2) \,\rrbracket_f &= \llbracket\, e_1 \,\rrbracket_f \llbracket\, e_2 \,\rrbracket_{f+1} \ \texttt{OP\_MIN}, \text{ or} \\
\llbracket\, max(e_1, e_2) \,\rrbracket_f &= \llbracket\, e_1 \,\rrbracket_f \llbracket\, e_2 \,\rrbracket_{f+1} \ \texttt{OP\_MIN},
\end{aligned}$$

then, since both computations on the right hand-side must be derived with less than n steps, each of them increase the pointers of the variable function by one. Because the opcodes following them will decrease the variable function by exactly one, the claim holds true.

The pointers of the variable function is therefore increased by exactly one in all cases. □

Logical expressions

Logical expressions perform operations returning `true` or `false`. As discussed above when the data expressions were introduced, it should also not be possible for logical expressions to compare hashes or data directly with arithmetic expressions, but will be taken care of with a type system.

$$\begin{array}{rl} l, l_1, l_2 := & x \mid true \mid false \mid e_1 == e_2 \mid e_1 ! = e_2 \mid \\ & e_1 < e_2 \mid e_1 > e_2 \mid e_1 <= e_2 \mid e_1 >= e_2 \mid within(e_1, e_2, e_3) \mid \\ & !l \mid l_1 \& l_2 \mid l_1 || l_2 \mid l_1 == l_2 \mid l_1 ! = l_2 \end{array}$$

Script offers two different opcodes to check for equality. One of them is `OP_NUMEQUAL`, which compares two values numerically. With `OP_NUMEQUAL` the values are evaluated to their smallest integer representation and are then compared to each other. It therefore compares integers for numerical identity. The other operation is the equivalence operation `OP_EQUAL`, and it compares two values bit-wise. Hence the number "1" represented as $0x01$ and $0x0001$ would not be equivalent, but still numerical identical.

Since numerical expressions are always used in their smallest integer representation, for NextScript it suffices to only use the equivalence operation.

$$\begin{array}{lcl}
[\![true]\!]_f & = & \texttt{OP\_1}, \\
[\![false]\!]_f & = & \texttt{OP\_0}, \\
[\![e_1 == e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_EQUAL}, \\
[\![e_1 ! = e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_EQUAL OP\_NOT}, \\
[\![e_1 < e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_LESSTHAN}, \\
[\![e_1 > e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_GREATERTHAN}, \\
[\![e_1 <= e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_LESSTHANOREQUAL}, \\
[\![e_1 >= e_2]\!]_f & = & [\![e_1]\!]_f [\![e_2]\!]_{f+1} \;\; \texttt{OP\_GREATERTHANOREQUAL}, \\
[\![within(e_1, e_2, e_3)]\!]_f & = & [\![e_1]\!]_f \; [\![e_2]\!]_{f+1} \; [\![e_3]\!]_{f+2} \;\; \texttt{OP\_WITHIN}, \\
[\![!l]\!]_f & = & [\![l]\!]_f \;\; \texttt{OP\_NOT}, \\
[\![l_1 \& l_2]\!]_f & = & [\![l_1]\!]_f [\![l_2]\!]_{f+1} \;\; \texttt{OP\_BOOLAND}, \\
[\![l_1 || l_2]\!]_f & = & [\![l_1]\!]_f [\![l_2]\!]_{f+1} \;\; \texttt{OP\_BOOLOR}, \\
[\![l_1 == l_2]\!]_f & = & [\![l_1]\!]_f [\![l_2]\!]_{f+1} \;\; \texttt{OP\_EQUAL}, \\
[\![l_1 ! = l_2]\!]_f & = & [\![l_1]\!]_f [\![l_2]\!]_{f+1} \;\; \texttt{OP\_EQUAL OP\_NOT},
\end{array}$$

Evaluation rules for expressions

The evaluation rules for expressions in NextScript are straightforward, and should be already clear from the names. The usual operator precedence is used. In the following a selection of evaluation rules are presented.

$$\frac{h \rightarrow v}{hash160(h) \;\rightarrow\; hash160(v)} \text{ E} - \text{HASH160}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 + e_2 \;\rightarrow\; v_1 + v_2} \text{ E} - \text{SUM}$$

$$\frac{e \rightarrow v}{e{+}{+} \;\rightarrow\; v + 1} \text{ E} - \text{INC}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 < v_2}{min(e_1, e_2) \;\rightarrow\; v_1} \text{ E} - \text{MIN1}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 \geq v_2}{min(e_1, e_2) \;\rightarrow\; v_2} \text{ E} - \text{MIN2}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 < v_2}{e_1 < e_2 \;\rightarrow\; true} \text{ E} - \text{LESS1}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 \geq v_2}{e_1 < e_2 \;\rightarrow\; false} \text{ E} - \text{LESS0}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 \neq v_2}{e_1! = e_2 \;\rightarrow\; true} \text{ E} - \text{NOTEQUAL1}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad v_1 = v_2}{e_1! = e_2 \;\rightarrow\; false} \text{ E} - \text{NOTEQUAL0}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad e_3 \rightarrow v_3 \quad v_1 \leq v_2 \leq v_3}{within(e_1, e_2, e_3) \;\rightarrow\; true} \text{ E} - \text{WITHIN1}$$

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad e_3 \rightarrow v_3 \quad v_1 > v_2 \lor v_3 < v_2}{within(e_1, e_2, e_3) \;\rightarrow\; false} \text{ E} - \text{WITHIN0}$$

7.3 Programs

We now turn to defining the syntax of a NextScript program by its grammar. Recall, that a Script program consists of two parts. A scriptSig, in which values are provided, and a scriptPubKey containing functions and other values. With the scriptSig prepended to the scriptPubKey, the

Script program must end with `true` left on the stack. Most values of a Script program will be delivered by the scriptSig. Those can often not be provided beforehand, as this would imply that the sender creating the output could himself spend the bitcoins again, by his capability to create a valid input to spend from the output. Signatures must for example typically be provided with a scriptSig.

Just like Script programs, a NextScript program is made up of two parts. The first part is the *input program*, which will compile to a scriptSig and delivers values. The other part is the NextScript *output program*, which can contain commands and values, and compiles to a scriptPubKey. When using NextScript, values that should be provided by an input program, must beforehand already be declared in some special way inside the output program. The output program is therefore divided into an input and output part. The input part contains only so called *input variables*, which define *how* the input program must deliver its values, in other words in which order they must be provided. The output part of the output program will contain commands and values, and corresponds to the actual scriptPubKey in Script. The compiler is the same for both input and output programs. But the grammars differ between them. In the following we will look at output programs and their available commands.

Output programs

In this section we will look at the grammar of output programs, and how the input and output parts are put together. But first we will see how variables inside output programs are declared to contain values.

Declaring variables

A variable in an output program can be declared with $let\,x = e$, or $let\,x = l$. When compiled, the variable function is updated to contain the altered pointers of the prevailing variables, and also a pointer of the new variable x. Since the value after declaring the variable x will be on the top of the stack, the variable function becomes $f_{+1}[x \mapsto 1]$. As will be discussed in the section regarding the scope, a variable will only be declared within the scope of the smallest clause it is contained in.

The compilation for declaring a variable is thus:

$$\begin{aligned} [\![\, let\,x = e; P \,]\!]_f &= [\![\, e \,]\!]_f \, [\![\, P \,]\!]_{f_{+1}[x \mapsto 1]} \\ [\![\, let\,x = l; P \,]\!]_f &= [\![\, l \,]\!]_f \, [\![\, P \,]\!]_{f_{+1}[x \mapsto 1]} \end{aligned}$$

It has already been shown, that after compiling an expression, the top stack item will contain the value for the expression and the variable function f is increased to become 1-free. The last step therefore simply sets the pointer of the new variable x to 1 inside f_{+1}. So the variable function is updated accordingly to become $f_{+1}[x \mapsto 1]$.

To update the value of a variable, that variable can simply be declared again with the new value. The variable function will then remove the old pointer of the variable and use pointer 1 instead, just as the compilation above suggests. As discussed previously though, the data of the "old" variable is not removed from the stack.

Declaring input variables

In output programs we must also be able to declare input variables, which cannot contain any data when the program is compiled. Input variables also allow to comprehend in an understandable and predictable way which data is needed to spend the transaction for which the output program is constructed.

Writing $input\; x : T_{in}$ will reserve the variable name x for a value, which must be provided by a scriptSig. The purpose of T_{in} inside the declaration of an input variable is to indicate what the type of values should be provided by an input program to be consistent with those expected by the output program. Every input variable must be declared in the beginning of a NextScript program. The border between the input and output part of a NextScript output program is exactly when the last input variable is declared and the first program function is called or a variable with value defined.

Values for input variables will first be delivered by an input program and the whole program is first then being executed. Compiling input variables does therefore not involve computing values, and hence no actual compilation to Script code is necessary. It solely involves updating the variable function in the same way if declaring a variable.

$$[\![\, input\; x : T_{in}; P \,]\!]_f \quad = \quad [\![\, \emptyset \,]\!]_f [\![\, P \,]\!]_{f_{+1}[x \mapsto 1]}$$

Grammar of output programs

Recall that a Script program must always end with the top element being `true`. But it can also be marked as invalid during execution and then terminate immediately. A command must therefore always evaluate to `true` or `false`, as it would not make sense for programs *solely* to encapsulate arithmetic or hash expressions. To define the grammar of an output program, we need to consider the possible NextScript commands that can be called inside a program. A NextScript program can be a composition of logical expressions and commands, which will be defined in the following. The logical expressions used outside of an if-statement are encapsulated inside a *check* function, which makes sure that the expression evaluates to `true`. Every other command must also evaluate to `true`. If a check function or command does not evaluate to `true`, the transaction is instantly marked as invalid. Its implementation will be later shown in the compilation rules. The commands take as arguments the expressions defined previously.

Some of them are expected to evaluate to e.g. a signature, public key, hash value, or values for locktime or the sequence number. Since it is not possible to e.g. *calculate* a public key using an expression, those values will typically simply be provided directly as terminal values or with variables containing that value. In the following the available commands are presented.

$$
\begin{aligned}
Q, Q_1, Q_2 := \;& let\, x = l;\; Q \quad | \quad let\, x = e;\; Q \quad | \quad check(l);\; Q \quad | \\
& sendTo(e_1, e_2, e_3);\; Q \quad | \\
& checkSig(e_1, e_2);\; Q \quad | \\
& checkMultiSig(e_1, \cdots, e_M \,|\, e'_1, \cdots, e'_N);\; Q \quad | \\
& checkLocktime(e);\; Q \quad | \\
& checkSequence(e);\; Q \quad | \\
& if(l)\,\{\, Q_1 \,\};\; Q \quad | \quad if(l)\,\{\, Q_1 \,\}\; else\,\{\, Q_2 \,\};\; Q \quad | \quad \epsilon
\end{aligned}
$$

- $check(l)$: Checks if the logical expression l returns `true`.

- $sendTo(e_1, e_2, e_3)$: Creates a P2PKH instruction.
  The value $e_1 \rightarrow v_{sig}$ is expected to be the signature to the corresponding public key $e_2 \rightarrow v_{pub}$. The value $e_3 \rightarrow v_{pubH}$ is the expected 20-bytes-hash value of the public key.

- $checkSig(e_1, e_2)$: Creates a P2PK instruction, which is outdated by the P2PKH from above. The value $e_1 \rightarrow v_{sig}$ is the signature to the corresponding public key $e_2 \rightarrow v_{pub}$.

- $checkMultiSig(e_1, \ldots, e_M \,|\, e'_1, \ldots, e'_N)$: Creates a multisignature instruction, which requires M out of N possible signatures.
  The signatures $e_1 \rightarrow v_{sig1}, \ldots, e_M \rightarrow v_{sigM}$ and the public keys $e'_1 \rightarrow v_{pub1}, \ldots, e'_N \rightarrow$

v_{pubN} are expected to be as described for the $checkSig$ function above. To distinguish between the expressions containing the signatures and public keys, the delimiter | is used. Furthermore, $M \leq 20$. But if used in a P2SH with *sendToScript*, then $M \leq 16$ due to the 520 bytes size limit of the push of the redeem script.

- $checkLocktime(e)$: Creates a locktime instruction that makes the output unspendable until the locktime has passed.
  The value $e \to v_{lock}$. Recall that if $0 \leq v_{lock} < 500.000.000$, then v_{lock} is regarded as a block height; if $500.000.000 \leq v_{lock} \leq 4.294.967.295$, then as a UNIX timestamp. Recall, that this opcode does not consume its operand. When the command is compiled, the operand is therefore popped explicitly.

- $checkSequence(e)$: Creates an instruction that prevents the output from being spend until a relative time after the transaction was confirmed has passed.
  The value $e \to v_{seq}$. The value v_{seq} is either time-based or block-based depending on a type bit set within the sequence number. Recall that this opcode does not consume its operand. When the command is compiled, the operand is therefore popped explicitly after the operation.

The compilation of the commands from above are shown below. The compilation for declaring variables was already shown before, and is therefore not repeated.

$[\![\, check(l) \,]\!]_f$	=	$[\![\, l \,]\!]_f$ `OP_VERIFY`
$[\![sendTo(e_1, e_2, e_3)]\!]_f$	=	$[\![\, e_1 \,]\!]_f$ $[\![\, e_2 \,]\!]_{f+1}$ `OP_DUP` `OP_HASH160` $[\![\, e_3 \,]\!]$ `OP_EQUALVERIFY` `OP_CHECKSIGVERIFY`
$[\![\, checkSig(e_1, e_2) \,]\!]_f$	=	$[\![\, e_1 \,]\!]_f$ $[\![\, e_2 \,]\!]_{f+1}$ $\texttt{OP\_CHECKSIGVERIFY}_{f+2}$
$[\![checkMultisig(e_1, \ldots, e_M, e'_1, \ldots, e'_N)]\!]_f$	=	`OP_0` $[\![\, e_1 \,]\!]_{f+1}$ ... $[\![\, e_M \,]\!]_{f+M}$ `<M>` $[\![\, e'_1 \,]\!]_{f+2+M}$ ... $[\![\, e'_N \,]\!]_{f+1+M+N}$ `<N>` `OP_CHECKMULTISIGVERIFY`
$[\![\, checkLocktime(e) \,]\!]_f$	=	$[\![\, e \,]\!]_f$ `OP_CHECKLOCKTIMEVERIFY` `OP_DROP`
$[\![\, checkSequence(e) \,]\!]_f$	=	$[\![\, e \,]\!]_f$ `OP_CHECKSEQUENCEVERIFY` `OP_DROP`

Every compilation of a command or check function makes sure that it evaluates to `true`.

This is simply done using the opcode OP_VERIFY. Take for example the function $sendTo(e_1, e_2, e_3)$, which produces a Pay-To-PubKey-Hash instruction. The first verification uses OP_EQUALVERIFY to check that the provided public key corresponds to the specified Bitcoin address. If instead OP_EQUAL would have been used, then no matter the outcome of OP_EQUAL, the execution would simply continue. In the final step, when a script program end, the top value of the stack is checked if it is true. But this value could potentially only be returned by exactly one command. Therefore it is important that each command in a program is checked after its execution by using OP_VERIFY, making sure it evaluates to true.

It may have raised some concern that in fact the compilation above exactly ignores that a valid program must nevertheless also end with true left on the stack! This issue will be addresses in the following, where the complete and final grammar of an output program is defined.

To define the complete grammar, we will also need to consider the possibilities of making different transaction types. Those transaction types were discussed previously. One such special type is the Pay-To-Script-Hash from Section 6.1, which is recognized based on its Script pattern. No other opcodes than the ones defined in its pattern are allowed to appear inside a Pay-To-ScriptHash transaction. The NextScript command to produce this kind of transaction is called *sendToScript*, and it will also mark the end of the program producing the redeem script, which for the Pay-To-Script-Hash transaction.

It is also possible to use a transaction output solely to store small chunks of data, without actually sending bitcoins. This is done with the opcode OP_RETURN. An output containing OP_RETURN is unspendable due to rules defined by the Bitcoin protocol. This transaction type should therefore only use the opcode OP_RETURN followed by the data to store. The program command is called *sendData*.

A NextScript output program P is then defined in the following, where D contains the input variables that must be declared prior to any other command in Q.

$$P := \quad D;\ Q;\ \texttt{OP\_TRUE};\ sendToScript \quad | \quad sendData \quad | \quad D;\ Q;\ \texttt{OP\_TRUE}$$

Before going into details with the commands of $sendToScript$ and $returnData$, we discuss why the opcode OP_TRUE appears in the grammar at the end of a program. In the grammar above, true is always pushed with the last operation onto the stack, and therefore the top value will always be `true`. The reason for this is exactly what was mentioned about the checks in Q. By the construction of the commands in Q, every command is immediately checked to evaluate to true after its execution. If any one of them does not, then the transaction is instantly marked as invalid and terminates. If a NextScript program therefore reaches to its end, it implies that all checks during its execution must have passed without failure, and hence it is valid and `true` should be pushed onto the stack to mark it as such.

The commands for $sendToScript$ and $returnData$ can be compiled as follows.

- $returnData(e)$: Creates an `OP_RETURN` transaction.
  This program will create a scriptPubKey that is unspendable. The data expressed in e is allowed to be of at most 520 bytes due to the push limit, but should preferable be of at most 80 bytes to comply with the "common practice" of using `OP_RETURN`.

- $sendToScript$: Creates a P2SH type of transaction.
  The so-called redeem program that appears before *sendToScript* can be of at most 520 bytes of size when compiled to Script. The size limit follows from the technical circumstance, that the compiled redeem program must be pushed as a whole onto the stack before being evaluated in Script, see 6.1. Its hash value is stored in `scriptHash`.

$$
\begin{array}{lcl}
[\![\, returnData(e) \,]\!]_f & = & \texttt{OP\_RETURN}\ \ [\![\, e \,]\!]_f \\
[\![\, sendToScript \,]\!]_f & = & \texttt{OP\_HASH160 <scriptHash>} \\
 & & \texttt{OP\_EQUAL}
\end{array}
$$

The input variables are simply defined by the following grammar, where x denotes the variable name that can be chosen freely. Recall that T denotes a type, which will be defined when introducing the type system.

$$D := \quad input\ x : T;\ D \quad | \quad \epsilon$$

Scope

The if/else-clauses of Q need some more consideration. As deciding which branch of an if/else-clause will be executed depends on the actual input values, all variables declared within it must be local to that clause, and may not be used outside of it. A *restore process* will therefore run immediately after a clause was executed. The same variable function used when entering the if-clause can then also be used again when leaving the clause. Due to the restore process, all pointers in the variable function will remain valid, both if the clause was executed or not. The variable function can therefore be used oblivious about execution of an if- or if/else-clause.

The restore process is a sequence of commands generated by a function called $restore$, which takes as input a variable function f. It will restore the pointers of the variable function f, to the state before entering the clause, by rearranging the values on the stack. Recall that there is no difference between declaring a variable and updating its value. The rearrangement is therefore done in such a way, that the positions to the updated values are stored at the same positions where the original values were, while the positions of unchanged values stay the same.

Technically, this rearrangement is done by using the simplest possible approach. The simplicity of understanding the process is paid for by bloating the Script program with a lot of push opcodes. A discussion about optimization will follow at the end of this chapter in Section 7.6.

Recall that Script only has opcodes to copy values onto the top of the stack. The $restore$ process will therefore copy to the top of the stack the values all those variables that were in the variable function f before entering the clause. This is done one-by-one in reversed order, so that the rearrangement will contain the correct ordering when the process ends.

In other words, the variable that has the "largest" pointer n in f will be simply copied to the top. Assuming there is another variable with the next pointer $n - 1$, then its value is copied to the top, otherwise a dummy value is pushed onto the stack. The process is continued likewise for the next pointer $n - 2$, until reaching pointer 1. The stack will now contain the values to the variables in exactly the same order and with the same pointers as before entering the clause. Copies of those values may still exist in some ordering at the bottom of the stack, but they will simply be ignored in the continuance of the execution.

More formally, the process is defined as follows:

$$restore(f) = \bigoplus_{i=n}^{1} \begin{cases} [\![\, x_i \,]\!] & \text{if } \exists\, x_i \text{ s.t. } f^{-1}(x_i) = i \\ \texttt{OP\_0} & \text{otherwise} \end{cases}$$

The symbol $\oplus$ denotes a concatenation, and the value of n is the largest pointer in f.

The compilation of an if-clause is then:

$$\begin{array}{rcl} [\![\, if(l)\ \{\, Q_1\,\}; Q\,]\!]_f & = & [\![\, l\,]\!]_f \\ & & \texttt{OP\_IF}\, [\![\, Q_1\,]\!]_f\, restore(f) \\ & & \texttt{OP\_ENDIF} \\ & & [\![\, Q\,]\!]_f \end{array}$$

Similarly for an if/else-clause:

$$\begin{array}{rcl} [\![\, if(l)\ \{\, Q_1\,\}\ else\ \{\, Q_2\,\}; Q\,]\!]_f & = & [\![\, l\,]\!]_f \\ & & \texttt{OP\_IF}\, [\![\, Q_1\,]\!]_f\, restore(f) \\ & & \texttt{OP\_ELSE}\, [\![\, Q_2\,]\!]_f\, restore(f) \\ & & \texttt{OP\_ENDIF} \\ & & [\![\, Q\,]\!]_f \end{array}$$

Procedures

Procedures are implemented as macros. They are effectively just regarded as a piece of code that when compiled will be substituted for every call of the procedure.

A procedure therefore does not directly "return" a value. Instead, parameters can be defined which either contain values, or variable names to which values can be stored. Formally, procedures are defined as follows:

$$\mathfrak{D} := \{function\, X_i(\bar{x}_i) = Q_i;\ \}$$

Compiling a procedure then simply substitutes it with the compiled code. Assume there is a call to a procedure $X(\widetilde{y}) = Q_1$, then the compilation is:

$$\begin{array}{l} X(\widetilde{y}) = Q_1 \in \mathfrak{D} \\ [\![\, function\, X(\bar{y});\ Q\,]\!]_f = [\![\, Q_1[{}^{\bar{y}}/_{\bar{x}}];\ Q\,]\!]_f \end{array}$$

An example would be to define a procedure, calculating e.g. the median in any list of three values and storing it in the variable m:

```
function median(m,x,y,z) =
var m=0;
  if(y<x & x<z) {
    var m = x;
  } if(x<y & y<z) {
    var m = y;
} if(x<z & z<y) {
   var m = z;
}
```

Listing 7.1: An example for a procedure.

Input programs

We will now look at how values can be declared for the input variables. This assignment of values to input variables takes as mentioned place in input programs. An input program is compiled to a scriptSig. The name for an input program is somewhat of an overstatement, since the only purpose of input programs is to provide values.

$$N := \quad provide\, i;\ N \quad | \quad \epsilon$$

The expression i could technically be any expression, as when declaring variables in the output programs. But for the input programs it does actually not make sense to provide expressions that e.g. calculate the sum of two numbers. Input programs themselves, cannot receive any inputs, or values, from outside. It is therefore easier and more efficient to instead just directly provide values such as a sum without calculations.

For that reason, it is better to restrict the expressions to values only. Furthermore, since a program may execute conditionally depending on the value of an input variable, some values can be ignored during the execution. For those cases `NULL` can be declared to an input variable. This indicates that its value will not matter for the specific input program provided. Technically though, instead of `NULL` any other value could be provided instead. We will see examples of this in Chapter 8.

$$i := \quad true \quad | \quad false \quad | \quad integer \quad | \quad “string“ \quad | \quad NULL$$

The input program is oblivious about the corresponding variable names it declares the values for. Which variable is assigned a value depends on the order in which those values are declared in the input program. In the discussion at the end of this chapter we will look at how an input program could easily be extended to allow the assignment of values directly to the same variable names, which are used in the output program.

Execution rules for commands

We know present execution rules for NextScript. In actuality, a NextScript output program will typically be compiled to a scriptPubKey, and this scriptPubKey will then be executed as a Script program together with a scriptSig, which could be a compiled NextScript input program. The execution of the resulting Script program uses the rules of Script presented in Chapter 6.

But in the following, execution rules are presented for NextScript output programs, assuming that an input program did provide the required values that may be used as expressions in the commands. This is useful to show that NextScript behaves as expected and in the same way as its compiled Script code.

Recall from the discussion regarding the checks that commands will either simply continue execution, or terminate and mark the transaction as invalid if the check does not validate. We will for the execution rules use a function σ, which is an assignment function that maps variables to their values. The only execution rule that maps values to the σ function is the one used to declare variables and is presented below. The evaluation rules for expressions are denoted as $\rightarrow_\sigma$.

$$\frac{e \rightarrow_\sigma v}{\sigma,\ var\, x = e;\ Q\ \rightarrow\ \sigma[x \mapsto v]\ Q}\ \mathrm{E-VAR}$$

The $check$ command is executed as follows:

$$\frac{l \rightarrow_\sigma true}{\sigma,\ check(l);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-CHECK1}$$

$$\frac{l \rightarrow_\sigma false}{\sigma,\ check(l);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-CHECK0}$$

For the following commands, we assume that expressions used as parameters to signatures evaluate to signatures denoted as v_{sig}, and similarly for public keys denoted v_{pub}. The transaction is marked as invalid if the signature is invalid, and hence not computed from the public key or the transaction.

$$\frac{\begin{array}{c} e_1 \rightarrow_\sigma v_{sig} \quad e_2 \rightarrow_\sigma v_{pub} \quad e_3 \rightarrow_\sigma v_{pubH} \\ v_{sig}, v_{pub}\ \text{valid}\ \wedge hash160(v_{pub}) = v_{pubH} \end{array}}{\sigma,\ sendTo(e_1, e_2, e_3);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-SENDTO1}$$

$$\frac{\begin{array}{c} e_1 \rightarrow_\sigma v_{sig} \quad e_2 \rightarrow_\sigma v_{pub} \quad e_3 \rightarrow_\sigma v_{pubH} \\ v_{sig}, v_{pub} \text{ not valid } \vee hash160(v_{pub}) \neq v_{pubH} \end{array}}{\sigma,\ sendTo(e_1, e_2, e_3);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-SENDTO0}$$

$$\frac{e_1 \rightarrow_\sigma v_{sig} \quad e_2 \rightarrow_\sigma v_{pub} \quad v_{sig}, v_{pub} \text{ valid}}{\sigma,\ checkSig(e_1, e_2);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-CHECKSIG1}$$

$$\frac{e_1 \rightarrow_\sigma v_{sig} \quad e_2 \rightarrow_\sigma v_{pub} \quad v_{sig}, v_{pub} \text{ not valid}}{\sigma,\ checkSig(e_1, e_2);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-CHECKSIG0}$$

$$\frac{\begin{array}{l} e_1 \rightarrow_\sigma v_{sig1} \quad \cdots \quad e_M \rightarrow_\sigma v_{sigM} \\ e'_1 \rightarrow_\sigma v_{pub1} \quad \cdots \quad e'_N \rightarrow_\sigma v_{pubN} \\ \text{all } v_{sig1}, \cdots, v_{sigM} \text{ are valid} \end{array}}{\sigma,\ checkMultiSig(e_1, \cdots, e_M \mid e'_1, \cdots, e'_N);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-CHECKMULTISIG1}$$

$$\frac{\begin{array}{l} e_1 \rightarrow_\sigma v_{sig1} \quad \cdots \quad e_M \rightarrow_\sigma v_{sigM} \\ e'_1 \rightarrow_\sigma v_{pub1} \quad \cdots \quad e'_N \rightarrow_\sigma v_{pubN} \\ \text{at least one of } v_{sig1}, \cdots, v_{sigM} \text{ is invalid} \end{array}}{\sigma,\ checkMultiSig(e_1, \cdots, e_M \mid e'_1, \cdots, e'_N);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-CHECKMULTISIG0}$$

For the commands $checkLocktime$ and $checkSequence$, the expression they take as parameters must evaluate to a value for locktime v_{lock} or sequence v_{seq}, respectively. If the actual locktime of the transaction, or sequence to the input, is of at least this value, and of the same type, then the commands are executed without failure. Otherwise, the transaction is marked as invalid.

$$\frac{e \rightarrow_\sigma v_{time} \quad v_{time} \text{ is valid}}{\sigma,\ checkLocktime(e);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-CHECKLOCKTIME1}$$

$$\frac{e \rightarrow_\sigma v_{time} \quad v_{time} \text{ is not valid}}{\sigma,\ checkLocktime(e);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-CHECKLOCKTIME0}$$

$$\frac{e \rightarrow_\sigma v_{seq} \quad v_{seq} \text{ is valid}}{\sigma,\ checkSequence(e);\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-CHECKSEQUENCE1}$$

$$\frac{e \rightarrow_{\sigma} v_{seq} \quad v_{seq} \text{ is not valid}}{\sigma,\ checkSequence(e);\ Q\ \rightarrow\ \sigma,\ \text{invalid tx.}}\ \mathrm{E-CHECKSEQUENCE0}$$

The rules for the if and if/else clauses are straightforward.

$$\frac{l \rightarrow_{\sigma} true}{\sigma,\ if(l)\,\{Q_1\};\ Q\ \rightarrow\ \sigma,\ Q_1;\ Q}\ \mathrm{E-IF1}$$

$$\frac{l \rightarrow_{\sigma} false}{\sigma,\ if(l)\,\{Q_1\};\ Q\ \rightarrow\ \sigma,\ Q}\ \mathrm{E-IF0}$$

$$\frac{l \rightarrow_{\sigma} true}{\sigma,\ if(l)\,\{Q_1\}\,else\,\{Q_2\};\ Q\ \rightarrow\ \sigma,\ Q_1;\ Q}\ \mathrm{E-IFELSE1}$$

$$\frac{l \rightarrow_{\sigma} false}{\sigma,\ if(l)\,\{Q_1\}\,else\,\{Q_2\};\ Q\ \rightarrow\ \sigma,\ Q_2;\ Q}\ \mathrm{E-IFELSE0}$$

7.4 Type system

As already mentioned, Script treats every value as hex data. Therefore Script cannot know when accessing a value, if it should be treated as an integer, a locktime or a public key. One of many issues this can cause is that of writing a program that accidentally treats a hash value as an integer, in the worst case causing a buffer overflow.

A type system contains a collection of rules to assign types to variables, expressions and commands. By examining the flow of a program, a type system can ensure that no type errors will occur. A type error could for example happen when comparing string data with numeric values, or performing arithmetic operations on strings. Type errors also include other operations that would not make sense, such as using a hash value as parameter for a command that expects a signature.

The type system therefore adds "meanings" to values and makes sure that the different parts of programs connects those values in a consistent way with regard to that underlying meaning.

The type system for NextScript uses a type environment Γ, and the "comma" operator, which extends Γ by adding a new binding on the right.[1] The empty context is denoted without a symbol, and it will be clear from the notation when being used.

The typing judgements for expressions are written as "$\Gamma \vdash e : T$" and "$\Gamma \vdash l : T$". Typing

[1]All variables declared with "$let\, x = e$" are distinct.

rules are defined by a set of inference rules assigning types to expressions. The typing relation is therefore the smallest ternary relation between contexts, expressions and types satisfying all instances of the rules. An expression e, or l, is *well typed* if there exist some Γ, T such that $\Gamma \vdash e : T$, or $\Gamma \vdash l : T$. Types assigned to expressions in output programs can be any of the following:

$T ::= Bool \quad | \quad Integer \quad | \quad Pubkey \quad | \quad String \quad | \quad Locktime \quad | \quad Sequence$

Note that there is no type for a "signature" in T. A signature should never be provided in the output program itself. It should instead be provided by an input program. Therefore the input variables in output programs can have the following types:

$T_{in} ::= Bool \quad | \quad Integer \quad | \quad Pubkey \quad | \quad String \quad | \quad Signature$

As can be seen from the definition of T_{in}, $Locktime$ and $Sequence$ should not be provided as input variables. The reason is that the spender of a transaction can anyways just set the locktime or sequence of his spending transaction to some desired value. There is no need to make an output program additionally check that some value the spender provides corresponds to the locktime or sequence of the transaction that the spender also already provides himself.

$Pubkey$ and $Signature$ are exactly like $String$. This distinction is a safety mechanism to prevent e.g. the accidental use of a hash value as a public key, for which no signature to unlock funds can possibly be provided. $Locktime$, on the other hand, is not exactly the same as $Integer$. The reason is that whereas arithmetic operations are limited to 4 byte signed integers, the locktime is represented as a 4 byte unsigned integer. Similarly for $Sequence$, but using a smaller range.

Typing rules for expressions

The typing rules for expressions in output programs are presented in the following. We begin by defining types for integers and Booleans:

$$\frac{}{\Gamma \vdash 0 : Integer} \; \mathrm{T-0}$$

and likewise for all 4 byte signed integers.

$$\frac{}{\Gamma \vdash true : Bool} \; \mathrm{T-TRUE}$$

$$\frac{}{\Gamma \vdash false : Bool} \text{ T} - \text{FALSE}$$

Likewise, $Locktime$ can be defined as

$$\frac{}{\Gamma \vdash 0 : Locktime} \text{ T} - \text{L0}$$

and likewise for all 4 byte unsigned integers.

Recall that the sequence number can be of two types, one which denotes a block number, and another which denotes a timespan in 512 seconds granularity, hence e.g. $10 = 5120$ seconds. The type $Sequence$ will only be considered of the type where it denotes a block number, and therefore its additional notion of a timespan in 512 seconds granularity is omitted.

$$\frac{}{\Gamma \vdash 0 : Sequence} \text{ T} - \text{S0}$$

up to the number 65.535, which is the maximum allowed sequence number and corresponds to roughly one year of a relative locktime.

$String$ is simply assigned to each subexpression h, hence

$$\frac{}{\Gamma \vdash h : String} \text{ T} - \text{STRING}$$

$$\frac{}{\Gamma \vdash h : Pubkey} \text{ T} - \text{PUBKEY}$$

Typing rules for expressions will be defined in the following, and are self explicable.

$$\frac{\Gamma \vdash h : String}{\Gamma \vdash \|h\| : Integer} \text{ T} - \text{SIZE}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 + e_2 : Integer} \text{ T} - \text{SUM}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 - e_2 : Integer} \text{ T} - \text{DIFFERENCE}$$

$$\frac{\Gamma \vdash e : Integer}{\Gamma \vdash |e| : Integer} \text{ T} - \text{ABS}$$

$$\frac{\Gamma \vdash e : Integer}{\Gamma \vdash e++ : Integer}\ \mathrm{T-INC}$$

$$\frac{\Gamma \vdash e : Integer}{\Gamma \vdash e-- : Integer}\ \mathrm{T-DEC}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash min(e_1, e_2) : Integer}\ \mathrm{T-MIN}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash max(e_1, e_2) : Integer}\ \mathrm{T-MAX}$$

For the logical expressions, some of the typing rules must be divided to make sure that expressions of the same types are compared. The types to consider are $T' ::= Bool \mid Integer \mid String$.

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1 == e_2 : Bool}\ \mathrm{T-EQUAL}$$

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1 != e_2 : Bool}\ \mathrm{T-NOTEQUAL}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 < e_2 : Bool}\ \mathrm{T-LESS}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 > e_2 : Bool}\ \mathrm{T-GREATER}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 <= e_2 : Bool}\ \mathrm{T-LESSEQUAL}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 >= e_2 : Bool}\ \mathrm{T-GREATEREQUAL}$$

$$\frac{\Gamma \vdash e : Bool}{\Gamma \vdash !e : Bool}\ \mathrm{T-NOT}$$

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 \& e_2 : Bool}\ \mathrm{T-AND}$$

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 || e_2 : Bool}\ \mathrm{T-OR}$$

$$\frac{\Gamma \vdash e_1 : Integer \quad \Gamma \vdash e_2 : Integer \quad \Gamma \vdash e_3}{\Gamma \vdash within(e_1, e_2, e_3) : Bool}\ \mathrm{T-WITHIN}$$

Variables and commands

Commands in output programs may take as parameters an expression e or l of types defined in T. The types can therefore be $Pubkey, String, Integer, Boolean, Locktime$ and $Sequence$.

When declaring a variable, it inherits the type of the expression it is assigned to:

$$\frac{\Gamma \vdash e : T \quad \Gamma, x : T \vdash Q}{\Gamma \vdash let\, x = e; Q}\ \mathrm{T-VARE}$$

$$\frac{\Gamma \vdash l : T \quad \Gamma, x : T \vdash Q}{\Gamma \vdash let\, x = l; Q}\ \mathrm{T-VARL}$$

Programs are not assigned to a type. But they are considered well typed if all of their parameters are well typed. The typing rules of programs are presented in the following.

$$\frac{\Gamma \vdash l : Bool \quad \Gamma \vdash Q}{\Gamma \vdash check(l); Q}\ \mathrm{T-CHECK}$$

$$\frac{\Gamma \vdash e_1 : Signature \quad \Gamma \vdash e_2 : Pubkey \quad \Gamma \vdash e_3 : String \quad \Gamma \vdash Q}{\Gamma \vdash sendTo(e_1, e_2, e_3); Q}\ \mathrm{T-SENDTO}$$

$$\frac{\Gamma \vdash e_1 : Signature \quad \Gamma \vdash e_2 : Pubkey \quad \Gamma \vdash Q}{\Gamma \vdash checkSig(e_1, e_2); Q}\ \mathrm{T-CHECKSIG}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash e_1 : \mathit{Signature} & \cdots & \Gamma \vdash e_M : \mathit{Signature} \\ \Gamma \vdash e'_1 : \mathit{Pubkey} & \cdots & \Gamma \vdash e'_N : \mathit{Pubkey} \end{array} \quad \Gamma \vdash Q}{\Gamma \vdash \mathit{checkMultiSig}(e_1, \cdots, e_M \mid e'_1, \cdots, e'_N); Q} \; \mathrm{T-CHECKMULTISIG}$$

$$\frac{\Gamma \vdash e : \mathit{Locktime} \quad \Gamma \vdash Q}{\Gamma \vdash \mathit{checkLocktime}(e); Q} \; \mathrm{T-CHECKLOCKTIME}$$

$$\frac{\Gamma \vdash e : \mathit{Sequence} \quad \Gamma \vdash Q}{\Gamma \vdash \mathit{checkSequence}(e); Q} \; \mathrm{T-CHECKSEQUENCE}$$

$$\frac{\Gamma \vdash l : \mathit{Bool} \quad \Gamma \vdash Q_1 \quad \Gamma \vdash Q}{\Gamma \vdash \mathit{if}(l)\,\{\,Q_1\,\};\; Q} \; \mathrm{T-IF}$$

$$\frac{\Gamma \vdash l : \mathit{Bool} \quad \Gamma \vdash Q_1 \quad \Gamma \vdash Q_2 \quad \Gamma \vdash Q}{\Gamma \vdash \mathit{if}(l)\,\{\,Q_1\,\}\,\mathit{else}\,\{\,Q_2\,\};\; Q} \; \mathrm{T-IFELSE}$$

Connecting input and output programs

Let us now return to the notion of input and output programs, how they are interconnected, and how the types of declaring input variables and providing values to them can be defined to match each other.

For the output programs we will add a layer that distinguishes between the part containing the scriptSig denoted D and the part containing the scriptPubKey denoted Q. For the latter we have that

$$\frac{\Gamma \vdash Q}{\Gamma \vdash Q : *}$$

where $*$ denotes that Q runs.

When combined with the input variables, the whole output program consists of a concatenation of variable declarations typed by

$$\frac{\Gamma, x : T_{in} \vdash P : T}{\Gamma \vdash \mathit{input}\, x : T_{in}; P : T_{in} \to T}$$

We can define the type of input programs with a similar concatenation. The empty input program is trivially typable.

$$\frac{}{\Gamma \vdash \epsilon : \top}$$

If the input program provides values, then their types can be concatenated:

$$\frac{\Gamma \vdash i : T_{in} \quad \Gamma \vdash I : T}{\Gamma \vdash provide\, i;\ I : T_{in} \wedge T}$$

Finally, the following lemma states that if an input program provides values of the same types as the output program declares in its input variables, then the program obtained by connecting them is well typed.

We will define $P[{}^{I}/_{D}]$ as the program defined by substituting the input variables of D inside program P with all of the provide commands in I.

Lemma. *If* $\vdash I : T_1 \wedge \cdots \wedge T_n$ *and* $\vdash P : T_1 \to \cdots \to T_n \to *$*, then* $\vdash P[{}^{I}/_{D}] : *$

Proof. By induction on I and P.
Base case:
$I : *$ and $P : *$. Then since there is no D in P, it follows immediately that $P[{}^{I}/_{D}] = P : *$.
Inductive step:
Assuming it holds for $I : T_1 \wedge \cdots \wedge T_n$ and $P = D; Q$, where $P : T_1 \to \cdots \to T_n$. If $P' = D'; Q'$, where $P' : T_1 \to \cdots \to T_{n+1}$ and $I' : T_1 \wedge \cdots \wedge T_{n+1}$, then

$$\frac{\vdash P'[{}^{I}/_{D}] : T_n}{P'[{}^{I}/_{D}] : T_n \vdash P'[{}^{I'}/_{D'}] : T_{n+1} \to *}$$

□

7.5 Theoretical results

In the following two important theoretical results regarding NextScript will be discussed. The first regards the type safety, which prevents erroneous programs. The second regards the compiler, and will be discussed afterwards.

Safety property

The *safety property* guarantees that a well typed program does not "go wrong", meaning it will never reach a stuck state that is not the end of the program, but from which also no further execution is defined.

To prove that the safety property holds true for NextScript, it suffices to prove two other properties in combination:

- *Progress*: A well typed command is not stuck. In other words, it can take a step according to evaluation rules.
- *Preservation*: If a well typed command takes a step of evaluation, then the resulting state of the program is also well typed.

For the properties to be described mathematically, we will define what it means for the assignment function σ to be well typed.

$$\frac{\forall x \in dom(\sigma) \quad \Gamma \vdash \sigma(x) : \Gamma(x)}{\Gamma \vdash \sigma}$$

where $\Gamma(x)$ is a function that returns the type of the variable x assigned in the environment Γ.

Property. *Progress: Assume Q is well typed, so $\Gamma \vdash Q$ and $\Gamma \vdash \sigma$. Then either $Q = \epsilon$, and we are done. Otherwise, it must be shown that for all possible assignments of σ in which Q is well typed, Q can make an execution step. Therefore,*
$\forall \sigma : \Gamma \vdash \sigma \quad \sigma, Q \rightarrow \sigma', Q'$

Proof. The proof is only sketched. Progress can be shown by induction on the cases for the execution rules. The only part that may not be directly clear is that some assignments of σ and Q can exist, which assign types that are required for the later evolution of Q to be well typed. □

Property. *Preservation: Assume Q is well typed and can take an execution step. Hence, $\Gamma \vdash \sigma$, $\Gamma \vdash Q$ and σ, $Q \rightarrow \sigma'$ Q'.*
Then the resulting state is also well typed, hence
$\Gamma' \vdash \sigma'$, $\Gamma' \vdash Q'$

Proof. The proof is only sketched. Preservation can again be shown by induction on the cases for execution rules. The only part that requires checking is that the resulting σ' is well typed. □

Soundness of the compiler

The soundness of the compiler regards the compilation of NextScript to Script. This property is useful, as it guarantees that both NextScript and Script behave similarly, that is, all possible behaviours are preserved, and no behaviour is added by NextScript. The compiled Script program therefore behaves according to the semantics of NextScript.

Property. *Soundness: If* $\sigma, Q \rightarrow \sigma', Q'$ *and* f *respects* σ*, then* $[\![Q]\!]_f \rightarrow [\![Q']\!]_{f'}$ *for some* f' *that respects* σ'*.*
By "f *respects* σ*" we mean that for all variables* x*, its value assigned in* σ *is stored at the position* $f(x)$ *on the value stack, and vice versa for* σ' *and* f'*.*

7.6 Optimisations

There are a number of possible ways to optimize the compiler. As mentioned earlier, the compiler can be considered optimal if the Script code it compiles to is as small in size as possible, hence uses as few opcodes and data pushes as possible.

The reason for this, is that a transaction fee in Bitcoin is typically paid for depending on its size. Currently a block is limited to not exceed 1 megabyte. The choice miners have of adding valid transactions into a new block therefore corresponds to a Knapsack Problem. Transactions paying the highest fee/size ratio will typically be included in a new block most rapidly.

But to give a guarantee of optimal compiled Script code size is out of reach for this thesis. In this section I will discuss and present improvements to the compiler, which generate Script code that will at least be smaller in specific situations. Although it adds complexity to the compiler, some of them are worth the consideration, and this section will end with a short discussion regarding this.

Optimizing the if/else statements

Improvements that need consideration regards the *restore* procedure after an if/else statement is executed. Recall, that the variable function used before entering a clause will also be used after execution of a statement, by making sure that the positions of values in the stack are restored.

But instead of restoring to the initial state, it suffices to make sure that oblivious of the execution of a clause, the variable function is the same, but possibly different from the initial one. This allows to use less copy operations, which the *restore* procedure performs on each of the variables.

Since there are a lot of points to consider, I will only briefly describe an alternative to the *restore* process, which will be called the *cleaning* process. If the *cleaning* is called after a simple if-clause that only declares local variables within its scope, then in fact popping all those values from the stack and returning to the initial variable function is still the most efficient approach.

Should the if-clause update a value of a variable outside the scope, then all the local variables that are declared after the last non-local variable was updated should be popped. By using the opcode `OP_NIP` one-by-one to pop each second-to-top element, this is then also possible for local variables that are directly preceding the last non-local one. An *else* branch could be introduced, which only copies the original values of updated variables onto the stack, and if necessary pushes dummy values before and after those copies that correspond to the local variables that were not popped from the *if*-branch.

Although, things can become more complicated if there are nested if-clauses involved. Then even popping the values of local variables that are further below in the stack could produce a smaller code, if otherwise all the corresponding else branches would each have to push dummy values.

If there are if/else statements involved, then to continue execution with the same variable function independently on which branch was executed the *cleaning* process must make sure that the branch with the fewest local variables pushes a corresponding number of dummy values. Those dummy values should simply be pushed to respect the order at which they appear as local variables in the other branch. Updates made to variables outside of the scope should be taken care of by copying the original value onto the top in the alternative branch, just as for the *else* statement that was introduced for the sole *if* statement.

Optimization of variables

There are two things than can be done to slightly optimize the compilation of variables, but I would consider the benefit of doing so limited. This should however be considered if one wishes that the compiled code to standard transaction types is optimal.

I will therefore informally just outline the idea. Specific push opcodes exist that can duplicate up to three values that are on the top, with a single opcode. Many commands need more than one input in form of e.g. variables. As it stands now, each variable is independently of the others compiled, such that the values are copied one-by-one onto the top of the stack.

If instead the variables were already in the order expected by some command, then they may be copied to the top together using a single opcode as described above. This however requires that the variables were already positioned at the top.

Furthermore, many Script programs are fairly small and simple. For the compiler to produce Script code of standard transaction types that is optimal, the provided variables that will not be used again should be directly consumed by the command. It may be possible to formalize a general rule, which allows variables to be consumed if it is decidable that they will not be used again afterwards. But this behaviour mostly benefits small programs and is therefore not further considered.

Optimization of input programs

An optimization can also be achieved by the order in which input variables are provided. This can be done in two places. First of all, output programs may be able to use fewer of the large copy opcodes `<n> OP_PICK`, if the most frequently used input variable would be initially provided at the top of the stack. Then simply using `OP_DUP` would suffice.

Furthermore, if we extend the discussion from before regarding the ability to consume variables that will not be used afterwards, then the values may be directly provided in the order in which they will be consumed by commands. This would require the compiler to analyze the output program and determine the best order of declaring variables, which may be difficult. But the compiler could simply check for all possible ways of declaring input variables and choose the one that produces the smallest compiled code. All possible orderings of input variables is the factorial of the number of input variables, and hence grows exponentially. The question remains open if this optimization is important enough for that.

Optimizing command calls

The last optimization I want to mention is in regard to patterns in which commands are called. I will show a program that uses $checkMultiSig$. I don't have any suggestion regarding a way to optimize the compiler to this regard, but to complete this section I nevertheless want to mention it.

```
input option;
input sig1;
input sig2;
input sig3;

if(option) {
  checkMultiSig(sig1, sig2, sig3 | "pubA", "pubB", "pubC");
} else {
  checkLocktime("now+3days");
  checkMultiSig(sig1, sig2 | "pubA", "pubB", "pubC");
}
```

Listing 7.2: An output program whose compilation could be optimized.

The program is shown in Listing 7.2. The compiled code calls `OP_CHECKMULTISIG` twice, and pushes the public keys "pubA", "pubB" and "pubC" twice, each of which is 30 bytes. But a Script code exists, which has the same functionality but is smaller.

The scriptPubKey that would be optimal is:

```
OP_IF OP_3
OP_ELSE OP_2
OP_ENDIF
<pubA> <pubB> <pubC> OP_3 OP_CHECKMULTISIG
```

So instead of pushing `OP_CHECKMULTISIG` together with the public keys twice, it suffices to simply decide if two or three signatures are required inside the if/else statement and specify the publkic keys outside the statement. This is a very specific examples, but more similar constructions can be made and it is possible to save on a lot of data pushes that way.

The importance of optimization

How important is optimization? This is a difficult question to answer. Bitcoin is still new and a lot can and will change. The transaction fee of a typical transaction of 300 bytes currently costs around 5 cents. Therefore getting rid of a single 1-byte opcode does not mean a lot. But saving a single push of a pubkey decreases the transaction size with around 10%. If programs were to become more complex, then potentially a lot could be improved regarding their sizes. The question is if transaction fees will stay low? Nobody knows. They could increase substantially in the future due to the limited size of blocks, which will be discussed in Chapter 8.

To conclude this section, I will draw parallels to the development of computer games. This may seem far away from Bitcoin, but I nevertheless believe they may share a common baseline regarding optimization in the beginning of new technologies. The first computer games were highly optimized. They were either completely written in Assembler or the compiled C code was optimized in Assembler. This allowed the development of games during the 1980s, which considering how little memory and computing there was available for that time, were very advanced. Nowadays memory and computation have become so big or fast, that optimization in this area does not matter as much anymore. I believe this could be the same path Bitcoin will take. I think in the short term, if more complex programs will arise in Bitcoin, then optimization will matter a lot. But optimization could slowly become less relevant in the future due to e.g. improvements on the bandwidth and possibilities of storage of the Blockchain. There are already a lot of improvements proposed for the Bitcoin protocol, and applications are developed that add another layer to steer around its limitations, which will be discussed in the following chapters.

Bitcoin is a technological tour-de-force.

Bill Gates

8 Applications and implementations

The transaction structure of Bitcoin was discussed in Chapter 4, and a formal model was presented in Chapter 6. In this chapter we will look at how the model can be combined with the language of NextScript presented in Chapter 7, to allow for complex transaction programs, used in different applications. Code examples of NextScript are presented to show how applications could be actually programmed.

Although a lot of applications can be made solely from the viewpoint of input and output programs, we need to consider the whole transaction structure in applications that make use of the different possible ways in which some programs can be validated.

The theoretical work regarding a formal model for the transaction structure has already been done, but time did not permit to also develop a practical framework that implements the model. Further investigation would therefore be required to make a platform, which can be used to implement complex transaction structures, as the ones presented in this chapter. But for the input and output programs, NextScript is used in all of the examples. The transaction structures are mathematically described using the formal model. It may be surprising how many complex constructions are possible with Bitcoin transactions, all of which Script is an important building block for.

When considering the transaction structure, the signature types become more important. For Script and NextScript, the difference between signature types does not matter. They simply check, if a provided signature is valid or not. There is no change in the execution depending on the signature types. But those signature types define how to construct a transaction in the first place.

The first examples in this chapter will show how the compiled Script code of a NextScript program will look like. But later examples, which also make use of the transaction structure to implement complex transactions, will only show the NextScript programs without the compiled Script code.

At the time of this writing, some of the later presented examples probably define the cutting edge of what the future might bring into the space, and demonstrate how flexible the scripting language is. It is our believe that NextScript makes it much easier to keep track of the process within a script and can improve the difficulty of programming such.

8.1 Basic examples

The following examples show a few simple output programs used in many standard transactions today. They should give a feeling for the programming language NextScript.

Pay-To-PubKey

The NextScript for a Pay-To-PubKey output can be considered the equivalence of a "*Hello World*" program. It is the most basic but still useful transaction type in Bitcoin, and is even of traditional value since the name for scriptSig and scriptPubKey roots in this type of transaction, see Section 6.1.

A Pay-To-PubKey can be programmed with the public keys directly specified as parameter for the $checkSig$ command, see Listing 8.1.

```
input sig:Signature;

checkSig(sig, "pubKey");
```

Listing 8.1: A Pay-To-PubKey program.

As defined by the grammar, the opcode `OP_TRUE` is added to the end of the program when compiled. The compiled code is the following scriptPubKey.

$[\![\, input\ sig : Signature \,]\!]$	$\emptyset$
$[\![\, checkSig(sig, \text{“}pubKey\text{“}) \,]\!]$	`OP_DUP`
	`<pubKey>`
	`OP_CHECKSIGVERIFY`
	`OP_TRUE`

This compiled scriptPubKey is slightly different from the Script code typically used as a Pay-To-PubKey. First of all, the value for the variable sig is first duplicated using `OP_DUP`. Furthermore, NextScript always uses the *verify*-type of opcodes where applicable, In the typical scriptPubKey instead `OP_CHECKSIG` is used, since it is the last check in the program. Finally the script ends with `OP_TRUE`, which could be omitted if `OP_CHECKSIG` would have been used instead. These are just minor differences, and many of which could be improved by optimizations discussed in Section 7.6. The differences are only in the opcodes, they of course do not change the behaviour of the script. Further discussion between the differences of the compiled code and how typically standard transactions are programmed is omitted in the following examples.

A NextScript input program, which validates the output program from above, must simply provide the signature:

```
provide "signature";
```

Listing 8.2: An input program for a Pay-To-PubKey.

Pay-To-PubKey-Hash

Now we will look at the most typical Bitcoin transaction being currently used. It is the Pay-To-PubKey-Hash that has been discussed in numerous occasions.

```
input sig:Signature;
input pub:Pubkey;

sendTo(sig, pub, "pubKeyHash");
```

Listing 8.3: A Pay-To-PubKeyHash program.

The compiled code is the following scriptPubKey.

$[\![\mathit{input\, sig} : \mathit{Signature}]\!]$	$\emptyset$
$[\![\mathit{input\, pub} : \mathit{Pubkey}]\!]$	$\emptyset$
$[\![\mathit{sendTo}(sig, pub, \text{“}pubKeyHash\text{“})]\!]$	`OP_OVER`
	`OP_OVER`
	`OP_DUP`
	`OP_HASH160`
	`<pubKeyHash>`
	`OP_EQUALVERIFY`
	`OP_CHECKSIGVERIFY`
	`OP_TRUE`

A NextScript input program that validates this output program must simply provide the signature and public key:

```
provide "signature";
provide "pubKey";
```

Listing 8.4: An input program for a Pay-To-PubKeyHash.

Zero-Knowledge Contingent Payment

A Zero-Knowledge Contingent Payment is a fancy name for a transaction type which resembles the hash-puzzle transaction we saw in Chapter 6. This transaction makes it possible to e.g. sell hash preimages in a secure manner, which means that if and only if the preimage is published will the transaction take place. An application to this is presented below, but will also be useful in regard to other applications discussed later.

A *zero-knowledge proof* (ZKP) for general computation is a cryptographic system which lets a person run an arbitrary program with a mixture of public and secret inputs and prove to others that this specific program accepted the inputs, without revealing anything more about its operation or the secret inputs.

Because these efficient ZKPs are cutting-edge technology, which depend on new strong cryptographic assumptions, their security is not settled yet. But in applications like ZKCP they are the only alternatives to third-party trust.

This field is a new research topic of itself, and out of scope for this thesis. Therefore no further details will be discussed about how they work. But accepting such a zero-knowledge proof system as a black box, the rest can be explained in simple terms.

Let's assume a buyer wants to buy a solution to a difficult problem, and finds a seller to that information. It could for example be a solution to a Travelling Salesman Problem. Then the buyer would perform a trusted setup for the zero-knowledge proof system and send the resulting setup information to the seller. The seller would pick a random encryption key y and encrypt the information the buyer wants to buy. Using the ZKP system the seller can construct a proof, which proves that

- E is an encryption of a solution to the buyer's problem.
- Y is the SHA-256 hash of the decryption key y for E.

The seller then sends E, Y, the proof and his public key $pubSeller$ to the buyer. The buyer can then verify the proof and conclude that if he learns the input to the SHA-256 function that yields Y, then he can decrypt the solution.

In other words, the buyer now only needs to buy the input y to the hash function, yields Y. There is a way to do that in Bitcoin! The buyer can create a payment to the output program shown in Listing 8.5.

```
input sigBuyer:Signature;
input sigSeller:Signature;
input y:String;

if(sha256(y)=="Y") {
  checkSig(sigSeller, "pubSeller");
} else {
  checkSequence(430);
  checkSig(sigBuyer, "pubBuyer");
}
```

Listing 8.5: An output program for a Zero-Knowledge Contigent Payment.

Either the seller collects the payment by providing his signature $sigSeller$ and y, which is the hash preimage of Y and what the buyer wants. Or, to avoid tying up the buyer's funds forever, if the seller does not collect the payment within some specified time, then the buyer can reclaim the payment after that. For the example we used $checkSequence$ with a relative locktime of 430 blocks, which roughly corresponds to three days. When the seller collects his payment, he is in other words forced to reveal the information y that the buyer needs in order to decrypt the answer.

8.2 Programmable transaction chains

Recall from Chapter 5 the notion of confirmation of transactions. A transaction is confirmed, if it is added to the blockchain. Before that happens, it could still be potentially double spent. An output is always completely spent by exactly one input of some spending transaction. But before confirmation in the blockchain, it is possible to create "potential" branches in the structure of a transaction chain, such that there could be different paths spending from an output. Ultimately the blockchain would only validate and contain one such path, regarding all the other paths as invalid double spend attempts.

In the following examples, recall that the structure of a transaction $t = (\widetilde{I}, \widetilde{O}, l)$, with any number of inputs $\widetilde{I} \ni I_i = (h_i, n_i, S_i, seq_i)$ and outputs $\widetilde{O} \ni O_j = (P_j, v_j)$. The values h_i and n_i denote the transaction hash and index of some output being spend with input I_i, S_i is the scriptSig, and seq_i is the sequence number. O_j is the j-th output, and P_j and v_j is the scriptPubKey and amount.

Micropayment channel

Complex constructions are possible by controlling which transactions can be chained together. One example is a so called *micropayment channel*. Sending a lot of transactions of only small amounts become very costly, see Figure 8.2.1. Transactions sending tiny amounts of bitcoins are also called *micropayments*, or *microtransactions*, and the motivation behind a micropayment channel is exactly to prevent from paying a lot of fees for them.

It is possible to construct an output from which potentially a lot of micropayments can be sent without paying a lot of fees. This is similar to a contract. When the contract terminates, a transaction that contains the right amount of balances will be signed and broadcasted to the network, and finally be confirmed by the blockchain. Only the fee of that last transaction would be necessary to pay, instead of for each single microtransaction.

The definition of a channel is rather broad and unspecific. But in the following we will

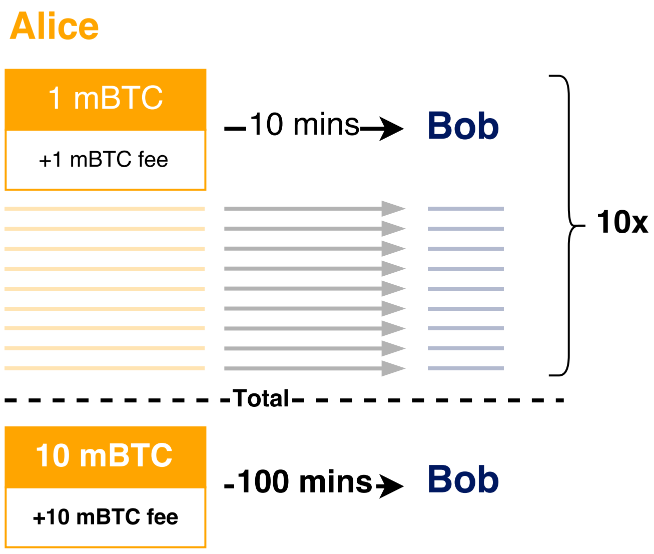


Figure 8.2.1: Schematic illustration of many *micropayments*. The last one is an aggregation of them all.

always refer to a channel as an output program that locks in an amount of bitcoins, whose function is to later send them to some other party, but without immediately settling the payment transactions on the blockchain. There will be intermediary states until reaching the final state triggered by the closing of the channel, which will settle the correct balances. Those intermediary states will only be known to the parties involved with the channel, while the Bitcoin network is oblivious about them and only knows about the final settlement transaction.

Funding transaction

Alice wants to open a channel with Bob. Within this channel she should be able to send a lot of micropayments to him. If Bob is uncooperative, she wants all of her money back after some beforehand agreed upon time, e.g. three days. This hook is Alice's security that if Bob simply disappears, her funds will not become potentially locked up forever in the channel.

A transaction that locks in some amount v in an output containing the program P_{fund}, could be setup as follows. The transaction is denoted as $t_{\text{fund}} = (\widetilde{I}, O, 0)$, where $\widetilde{I}$ are some inputs from Alice. The output $O = (P_{\text{fund}}, v)$ contains the NextScript program P_{fund} shown in Listing 8.6.

Let us assume that Alice creates a funding transaction t_{fund}, which is funded by e.g. 0.1 bitcoins by the input $\widetilde{I}$. In general, Alice would choose to lock in an amount, which she sup-

```
input sigA:Signature;
input sigB:Signature;
input option:Boolean;

if(option){
checkMultiSig(sigA, sigB, "pubKeyA", "pubKeyB");
} else {
checkLocktime("now+3days");
checkSig(sigA, "pubKeyA");
}
```

Listing 8.6: P_{fund}: The program for opening a micropayment channel. A denotes Alice and B denotes Bob. "$now + 3days$" inside $checkLocktime$ is simply used symbolically and should in actuality denote some absolut block height.

poses exceeds the total amount of micropayments she will send to Bob. But if necessary, she can always open a new channel to lock in additional funds. Before proceeding, she would have to broadcast this transaction to the Bitcoin network and have it confirm inside the blockchain. This would function as opening the channel. Only then can Bob accept the micropayments in a trust less matter, without the need to have them included into the blockchain immediately. How such micropayments could be constructed in detail will be regarded later.

Refund transaction

Let us first see how Alice could get her money back if Bob becomes uncooperative. In the output program P_{fund}, the function $checkLocktime$ has the parameter "$now+3days$". This is only used symbolically, and should actually contain the current block height of the blockchain with 430 blocks added to it. Since each block takes on average 10 minutes to mined, adding 430 blocks to the current blockchain height corresponds to an expected time of three days. During those three days, the channel would stay active, and Alice can by spending from this channel output continuously construct new payment transactions to Bob. In case Bob becomes uncooperative, Alice can wait for a total of three days to pass and get all of her money back from a refund transaction, whose locktime would be set to at least the value specified in the output program P_{fund}. The refund transaction is $t_{\text{refund}} = (I_{\text{A}}, O_1, \text{“}now + 3days\text{“})$, where the input $I_{\text{A}} = (h_{\text{fund}}, 0, S_{\text{refund}}, 0)$ spends the single output from t_{fund}. The NextScript input program S_{refund} would have to provide her signature with the ”option” value set to `false`, so that ”her”

```
provide "<Alice's signature>";
provide NULL;
provide false;
```

Listing 8.7: S_{refund}: The input program that refunds Alice. Note that if the last value corresponding to the option is set to true, then the value NULL should instead contain Bob's signature.

```
input sig:Signature;
input pub:Pubkey;
var pubHash = "Bob's address";
sendTo(sig, pub, pubHash);
```

Listing 8.8: P_1 simply sends the money to Bob's address. Similarly for P_2 but using Alice's address instead.

branch of the output program is executed. See Listing 8.7. The output O_1 would typically just send the money to Alice's Bitcoin address.

Payment transactions

With the security setup in place, let us now see how Alice within this channel actually can send e.g. 0.001 bitcoins to Bob. She creates a payment transaction t_{pay1} that contains two outputs O_1 and O_2. One of them must send 0.001 bitcoins to Bob's address, and the other one returns the rest of 0.098 bitcoins to herself. The difference in the amounts between the two outputs is 0.001 bitcoins and corresponds to the transaction fee. With the input I_{fund}, Alice spends from the channel. Whereas in the refund transaction Alice did provide a locktime to allow her to spend from the input without the need of Bob's signature, the locktime for the payment transaction is simply deactived by setting the value to 0. She can therefore not provide a valid input program, as it would need both hers and Bob's signature. Formally she constructs a payment transaction $t_{pay1} = (I_{\text{AB}}, \widetilde{O}, 0)$, where $I_{\text{AB}} = (h_{\text{fund}}, 0, \emptyset, 0)$, and $\widetilde{O} = (O_1, O_2)$ with $O_1 = (P_1, 0.001)$ and $O_2 = (P_2, 0.098)$. The NextScript output programs P_1 and P_2 must simply send the bitcoins to Alice and Bob. More importantly, note that the input program to I_{AB} at this point is empty. Alice signs this transaction structure t_{pay1} and gives both the structure and her generated signature to Bob. See Figure 8.2.2 for a schematic representation of the micropayment channel.

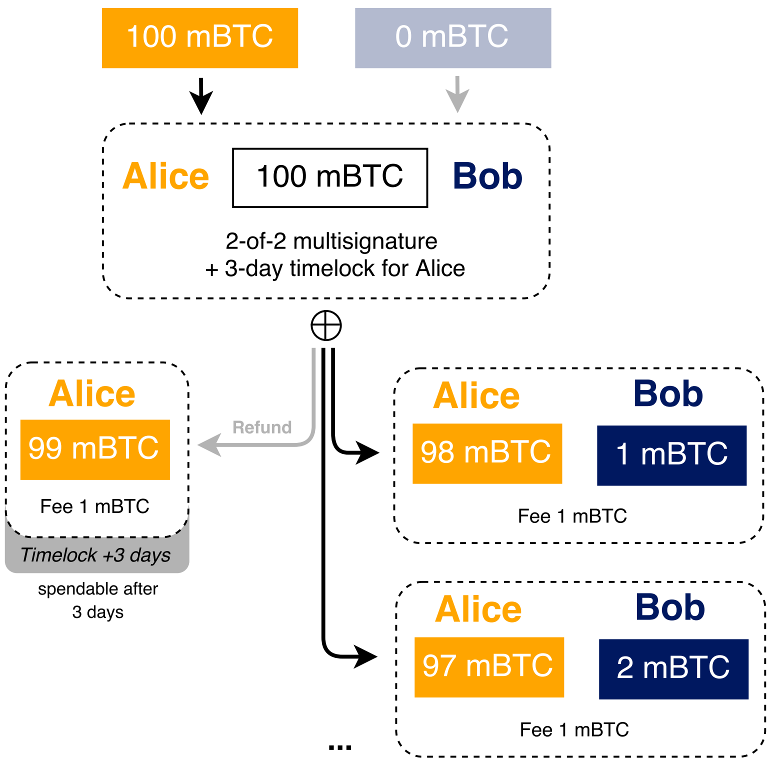


Figure 8.2.2: Showing the construction of a micropayment channel. On the right are payment transactions to Bob shown. On the left is the refund transaction that Alice can use after three days to refund her money if Bob should become uncooperative.

Bob could now also sign the payment transaction t_{pay1}, and could provide a valid input script from his and Alice's signature. But for now he will just keep the transaction as a security, knowing with certainty that during the next three days, at any time of his liking, he can broadcast t_{pay1} to receive the micropayment.

Payment transactions represent the balance of funds between Alice and Bob within the channel. A payment transaction is therefore not an additional payment per se, but instead incorporates also all previous payments made in this channel. If Alice now wants to send another micropayment to Bob, she would produce a new partially signed payment transaction t_{pay2} in nearly the same way as before and give it to Bob. But it should contain the total aggregated amount that Alice is sending to Bob within this channel. If Alice wants to send an additional 0.001 bitcoins to Bob, she would create a transaction that again spends from the funding output in t_{fund} as before, but this time sends 0.002 bitcoins to Bob and the rest of 0.097 bitcoins to herself, minus a transaction fee of 0.001 bitcoins. So, $t_{pay2} = (I_{\text{AB}}, \widetilde{O'}, 0)$ with I_{AB} spending from

t_{fund} as before and $\widetilde{O'} = (O'_1, O'_2)$ where $O'_1 = (P_1, 0.002)$ and $O'_2 = (P_2, 0.097)$. Since Bob would again be in possession of both signatures for this transaction, he can now provide valid input scripts to both transactions t_{pay1} and t_{pay2} C1a and C1b, respectively. But since they spend from the same output in t_{fund}, only one of them can be confirmed in the blockchain. Bob will therefore always choose the last payment transaction, since it is the one that pays him the most. It is also the one that reflects the correct balance between Alice and Bob inside this channel. All of the previous payment transactions can simply be discarded, but it is Bob's responsibility to e.g. not accidentally broadcast an old payment transaction to the Bitcoin network, Bob should also close the channel before the locktime specified in P_{fund} expires. He simply broadcasts his last received payment transaction by providing an input program as follows:

```
provide "<Alice's signature>";
provide "<Bob's signature>";
provide true;
```

The problem is that Alice can only send payments to Bob. The other way around does not work with this setup, since Bob and Alice will both favour a transaction, which sends themselves the most money, even if it would reflect an old state of balances. But this can be worked around with more sophisticated techniques, and are presented in the following section.

Lightning Network

Lightning Network provides the possibility to send bitcoins without the necessity to broadcast the transaction to the Bitcoin network. This may sound paradoxal, but can be done without trust in any other participant, even through means of different intermediary nodes. It is an example of how flexible the possibilities are solely by using Script and the transaction model.

Scalability issues

The idea behind the Lightning Network came up in regard to the scalability of the Bitcoin network, a topic that will be discussed briefly in Chapter 9. Just as for the micropayment channel, not all transactions of the Lightning Network have to be broadcasted to the Bitcoin network. This saves bandwidth and allows for a lot of transactions to happen even though the network capacity is at the time of this writing limited to about three transactions per second. By opening a single channel, Alice and Bob can send each other bitcoins forth and back without the need for transactions to actually confirm in the blockchain or paying a transaction fee. The trust less security is guaranteed by the ability for any one of them to close the channel at any time, independently on the other party. This would settle the current state of their balances on the blockchain, implicitly returning the correct amount to each of them. If one of them would try to cheat by broadcasting an old state of balances, this party will be maximally penalized by losing all their money put into this channel to the other party.

There exist different ways of realizing this idea. The most efficient one is presented and relies on the *immalleability* of transaction hashes. This is not yet possible with Bitcoin, but a change to the protocol that is being introduced the time of this writing, would make everything presented in the following directly possible. The change that is needed will be outlined in the following when applicable, and is discussed with more details in Chapter 9. The assumption of immalleability is, that a transaction hash, and hence the id, cannot even be changed for transactions that are not yet confirmed in the blockchain. Immalleability can be introduced by assuming that the scriptSig part of a transaction is not part of its hash value. This is assumed in the following constructions.

This assumption allows to reliably construct whole chains of transactions originating from some unconfirmed outputs in the blockchain, as long as the transactions in the chain cannot be double spent before eventually entering the blockchain.

For simplicity we will in the description below always omit adding a transaction fee, although they should normally always be included.

Funding transaction

Alice and Bob create in the following way a funding transaction to open a channel between each other in the Lightning Network. Let us assume that each party wants to lock 0.5 bitcoins in the channel. Then the funding transaction contains two inputs in which each contributes with their 0.5 bitcoins. The output to the funding transaction is a single 2-of-2 multisignature output, so it can only be spend by a transaction that contains both their signatures in the input program. The funding transaction can be modelled as $t_{\text{fund}} = (\widetilde{I}, O, 0)$, where $\widetilde{I}$ are the inputs of Alice and Bob containing e.g. 0.5 bitcoins each, and the output $O = (P_{\text{fund}}, 1)$ sends the whole one bitcoin to the program P_{fund}, which is simply a 2-of-2 multisignature, as seen in Listing 8.9.

```
input sigA:Signature;
input sigB:Signature;
checkMultiSig(sigA, sigB | "pubA", "pubB");
```

Listing 8.9: P_{fund} simply creates a 2-of-2 multisignature between Alice and Bob.

Commitment transactions

The output program P_{fund} of the funding transaction is different to the micropayment channel, as the funding transaction does not directly contain a hook for Alice or Bob to get a refund if the other party should become uncooperative. Therefore, before they both sign and broadcast the funding transaction t_{fund}, they each also create their version of a so called commitment transaction, denoted as C1a and C1b for Alice and Bob respectively.

Both C1a and C1b spend from the funding transaction t_{fund}. But as has been pointed out previously, we will assume that the hash of the funding transaction t_{fund}, which C1a and C1b are referencing inside their inputs, can be calculated, even though t_{fund} does not yet contain its input programs. Another important thing to notice is that both C1a and C1b spend from the same output, and therefore only one of them can become valid in the blockchain. But since C1a and C1b are distinguishable and only Alice (Bob) can broadcast C1a (C1b), this will later turn out to be used to prove which party did close the channel.

The commitment transactions C1a and C1b will each contain two outputs. C1a, which Alice creates, contains one output that directly pays Bob back his 0.5 bitcoins. The other output is of 0.5 bitcoins with a 2-of-2 multisignature between Alice and Bob and a relative locktime requirement to give Alice sole control from some time after the transaction was confirmed. This last output is similar to the output of a micropayment channel, but uses the sequence number as a relative relative locktime.

The commitment transactions C1a and C1b must now be exchanged. The other party must then partially sign and return it again, making it possible for the other party to create a valid input program to their commitment transactions by providing both signatures. The commitment transactions now act as security against the other party becoming uncooperative, in which case the commitment transaction can be broadcasted to the network and after confirmation would return the money. Figure **??** shows the construction so far.

Formally, $C1a = (I_{\text{fund}}, \widetilde{O}, 0)$, where input $I_{\text{fund}} = (h_{\text{fund}}, 0, S_{\text{fund}}, 0)$ references the funding transaction t_{fund} and the input program S_{fund} simply provides both Alice's and Bob's

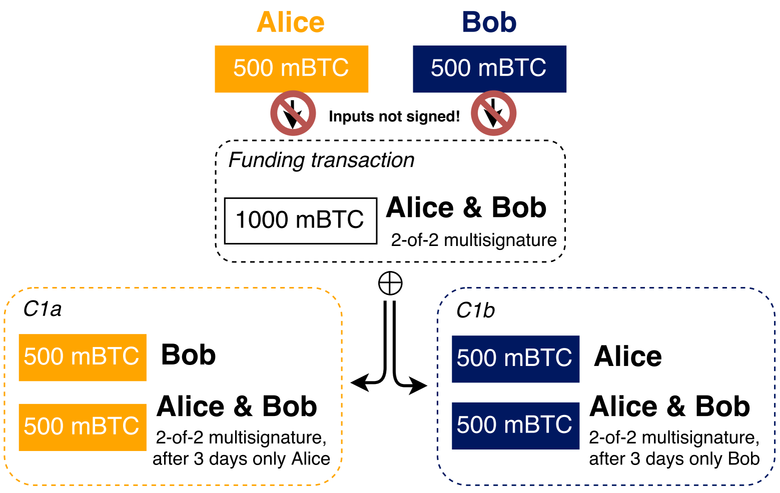


Figure 8.2.3: Alice and Bob fund a funding transaction, but do not provide valid input programs yet. Commitment transactions C1a and C1b are constructed that spend from the funding transaction and could settle the balances. Alice and only broadcast C1a, and Bob only C1b, deoted by the colouring.

signature. The outputs $\widetilde{O} = (O_1, O_2)$, with O_2 containing a *locktime multisignature* output program P_2, which can be seen in Listing 8.10, and is similar to a micropayment channel.

Revocable Delivery transactions

If the commitment transaction C1a were to be broadcasted and confirmed, then through the first output O_1 Bob would immediately receive his funds back. But for Alice to get her 0.5 bitcoins back, she would need Bobs approval on any transaction spending from the second

```
input sigA:Signature;
input sigB:Signature;
input option:Boolean;
if(option) {
  checkMultiSig(sigA, sigB | "pubA", "pubB");
} else {
  checkSequence(430);
  checkSig(sigA, "pubA")
}
```

Listing 8.10: P_2 creates a 2-of-2 multisignature between Alice and Bob with a relative timelock giving Alice sole control of the output after that time.

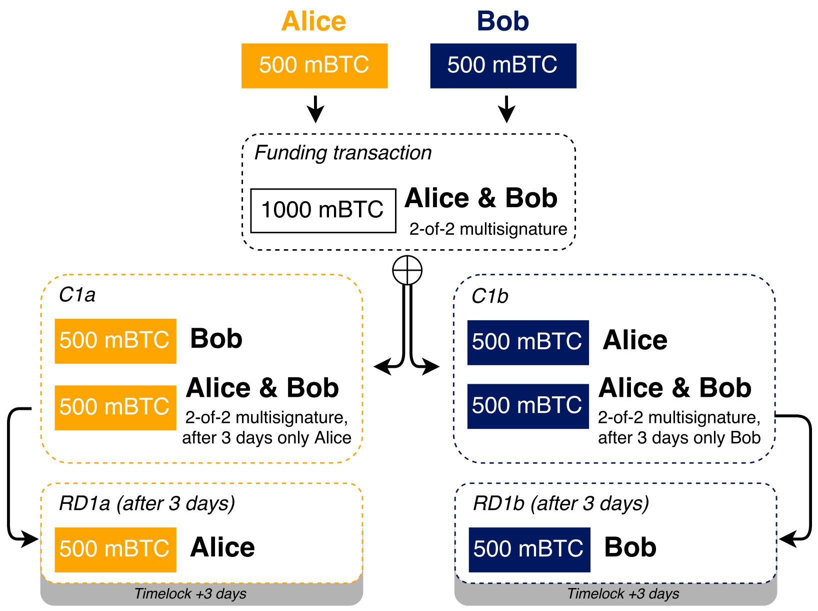


Figure 8.2.4: If any party broadcast their commitment transaction, then the other party immediately received their money from its first output. After the relative locktime (three days) have passed, the Revocable Delivery transaction could be used without cooperation from the other party to receive the funds that are tied up in the locktimed multisignature output.

output O_2 of C1a, by providing an input program with both hers and Bobs signature. But if the relative locktime was set to e.g. 430 blocks, then Alice can without Bobs cooperation spend from this output of C1a after it has reached maturity, solely by providing her own signature. This transaction will be referred to as a *Revocable Delivery transaction*, RD, and is similar to the refund transaction of a micropayment channel.

Any party has with C1a, respectively C1b, the ability to close the channel, and allow the other party to immediately receive their current amount of money. The party that closes the channel would have to wait until the relative locktime is expired, after which their RD transaction could be used to receive their money. Normally though, the other party would be cooperative and sign a spending transaction before the locktime has expired, so that the other party does not have to wait until using their RD transaction. The other party cannot steal any money, since it would require both signatures. The construction can be seen in Figure 8.2.4. Since the programs for RD transactions are very similar to a refund transaction in a micropayment channel, they are not further specified.

Breach Remedy transaction

What if Alice wants to send 0.1 bitcoins to Bob within this channel? Then they create a new pair of commitment transactions to reflect the updated balances. Alice creates a commitment transaction C2a, where now one output would send 0.6 bitcoins immediately to Bob, and the other output of 0.4 bitcoins is again a multisignature output with a relative locktime after which Alice solely controls that output. Bob does the same, but with an output paying 0.4 bitcoins directly to Alice and instead locking his 0.6 bitcoins into a relative locktime multisignature output. Just as before, they exchange those unsigned commitment transactions to have them returned by the other party partially signed. The new commitment transactions now reflect the current balances.

But the old commitment transactions reflecting old balances could still become confirmed when broadcasted to the Bitcoin network. Alice and Bob therefore also need to provide some kind of guarantee to never broadcast a previous commitment transaction. Each party creates and partially signs a so called *Breach Remedy transaction*, BR1a and BR1b for Alice and Bob respectively, and sends them to the other party. The BR transactions spend from the relative timelocked multisignature output of their commitment transaction. But since they contain both signatures in their input program, they would become immediately confirmed after the commitment transaction was confirmed, and would not have to wait for maturity as was the case for the RD transactions.

If Alice or Bob would broadcast a deprecated commitment transaction, that party would be penalized by losing all their money from the other party using their BR transaction to obtain it. This can be seen in Figure 8.2.5. The BR transaction is similar to a payment transaction within a micropayment channel.

As a technical side note, the public keys which Alice and Bob use in the multisignature outputs must be different for every new commitment transaction they create. If Bob now wants to send 0.1 bitcoins back to Alice, then the balance must change back to 0.5 bitcoins to each of them. They just create new commitment transactions C3a and C3b reflecting this balance, and the locktimed multisignature output will use keys that are different from any previous commitment transactions. Therefore the hash of every commitment transaction differs and the exchanged Breach Remedy transactions can only be used for the commitment transactions they are intended for.

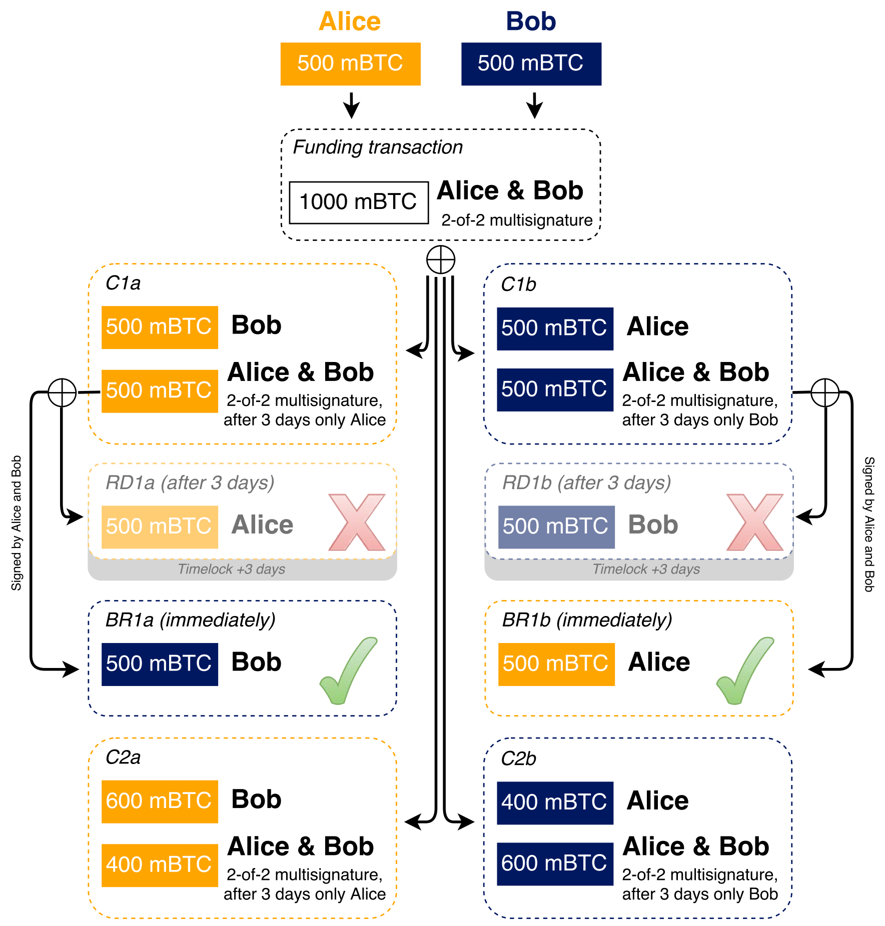


Figure 8.2.5: By signing the other parties Breach Remedy transaction, they agree upon to never broadcast a previous commitment transaction. The construction shows the state in which the balances are in respect to the commitment transaction C2a and C2b. The colour indicates which party can broadcast what transactions. If any one should broadcast an old commitment transaction C1a or C1b, then the other party can immediately receive all funds from their Breach Remedy transaction.

Routing transactions

The construction so far works for exactly two parties who can interact with each other to both send and receive bitcoins within the channel, without broadcasting each transaction to the Bitcoin network. But what if Alice wants to send money to Chuck, but Chuck and Alice do not have an open channel with each other? We would like for channel transactions to be able to be routed through the network. If Bob for example already has a channel with Chuck, then Alice could pay Bob and Bob could pay Chuck, instead of requiring Alice and Charlie to have an open channel together. The difficulty lies in doing this in a trust less manner, and solely relying on the Blockchain as the final "mediator".

The construction is extended with *Hashed Timelock Contracts* (HTLCs). They are in fact similar to the hash-based puzzle transactions discussed in Chapter 6 and the Zero Knowledge Contingent payments we saw in the beginning of this chapter. The purpose of a HTLC is to allow for global state across multiple nodes via hashes. This global state is ensured by time commitments and time-based unencumbering of resources via disclosure of preimages. Transactional "locking" occurs globally via commitments. And at any point in time, a single party is responsible for disclosing to the next party whether they have knowledge of some preimage. This construction does neither require trust in one's counterparty, nor any other party in the network. If something goes wrong the HTLC can be revoked.

A HTLC is a program in an additional output of a commitment transaction. It is similar to the output program of a ZKP. The first two outputs in a commitment transaction reflect the balances of each party in the channel. We already saw how sending bitcoins within this channel corresponds to updating the balances, reflected by chainging the output values of the new pair of commitment transactions accordingly. Now if Alice wants to route bitcoins through Bob, they would have to construct a new pair of commitment transactions, of which the first output reflects the unchanged balance of Bob, and the second depending on the amount Alice wants to route through Bob contains the changed balance of Alice. The programs used for these two outputs are just as described previously for commitment transactions. But a third, new output in the commitment transactions will additionally contain the amount Alice wants to route. It is locked to the HTLC, and for Bob to spend this output, he will have to provide a proof that he is forwarding Alice's payment to the next intermediary in the path.

The high-level idea behind it will be presented in the following. Assume that Alice wants to pay 0.1 bitcoins to Chuck through the intermediary Bob, that has an open channel with

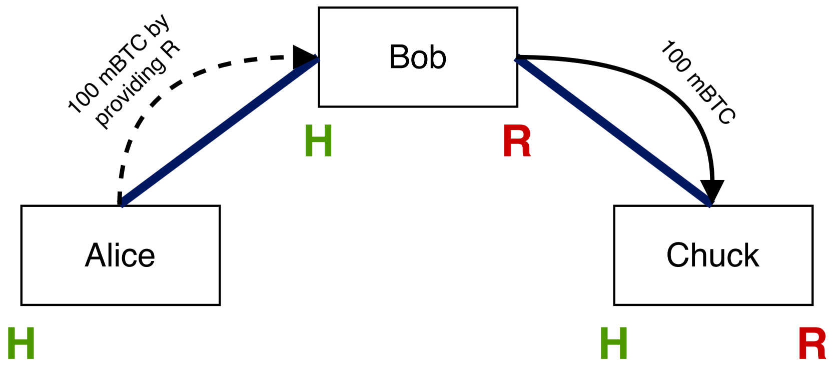


Figure 8.2.6: Illustration of the routing of transactions in the Lightning Network. Blue lines represent open channels between parties. Dotted arrows denotes HTLC payments, and straight arrows regular payments inside the channel.

both Alice and Chuck. In fact, there could be any number of intermediaries between Alice and Chuck. Chuck creates a random number R and hashes it, which results in H. He then gives H to Alice, but keeps R private. Then Alice creates a contract within the channel she has with Bob, paying him 0.1 bitcoins if the knows R, the preimage of H. Bob will then create a similar contract inside the channel he has with Chuck (or the next intermediary), paying 0.1 bitcoins to Chuck if he knows R, and Chuck knows R.

If Chuck tries to obtain the money by broadcasting the payment contract, he must reveal R. This mechanism is embedded into the output script of that contract, just like in a hash-based puzzle transaction or Zero-Knowledge Contingent Payment. And since R must be revealed, all the other payments that are conditional on R can also happen. See Figure 8.2.6 for a simple representation of routing transactions.

Formally, the new pair of commitment transactions could in continuation of our example for Alice and Bob be denoted as $C4a$ and $C4b$. They contain the following structure $C4a = (I_{\text{fund}}, \widetilde{O}, 0)$, where I_{fund} refers to the funding transaction, just as for all the other commitment transactions. The output $\widetilde{O}$ contains three outputs. O_1 reflects Bob's unchanged balance, and O_2 contains Alice's balance in accordance to the payment she wants to route. Recall that O_1 pays Bob immediately, while O_2 contains the locktimed multisignature for which Alice can obtain her money with a RD transaction after the relative locktime has passed. So far everything is as before. But the third output O_3 contains the HTLC, which sends the amount to be routed. Let $O_3 = (P_3, 0.1)$ in this example. The output program P_3 is similar to the Zero-Knowledge

```
input sigA:Signature;
input sigB:Signature;
input R:String;
if(sha256(R)=="H") {
  checkSig(sigB, "pubB");
} else {
  checkSequence(430);
  checkSig(sigA, "pubA")
}
```

Listing 8.11: P_3 creates the HTLC output.

Contingen Payment discussed earlier, just with a relative locktime. See Listing 8.11.

A similar HTLC output is constructed inside the channel of Bob and Chuck. After that point, Chuck is the only one who knows R. But Bob knows that as soon as Chuck would spend the HTLC output, Chuck would have to reveal R. This would immediately allow Bob to spend from the HTLC output he has with A, and hence in no way lose any money. There is one detail regarding the involved relative locktimes used inside each HTLC. TO not allow for timing attacks to become possible, the locktime requirements inside each HTLC should decrease between each intermediary. But this will not be discussed further.

But instead of broadcasting the commitment transaction, Chuck can just tell Bob what R is. To avoid broadcasting any transaction, Bob would after knowing R be willing to send that payment to Chuck within his channel without further conditions. This can be done in the usual way by creating commitment transactions that now reflect those balances without the use of an HTLC output. To avoid broadcasting any transaction onto the blockchain, Bob would reveal R to Alice and she would feel confident in now updating the balance in the channel with Bob in the same way, since the knowledge of R proves to Alice that Chuck has received the money by revealing it. So after R was revealed to every party, the states can simply be cleared out to not keep track of all potential new routing of payments. If any party should become unresponsive, the commitment transaction could be broadcasted; and the payment would either be forwarded through the knowledge of R, or it terminates and the money could be returned due to the relative locktime.

Nevertheless, since the liquidity is bounded by the total amount locked in a channel, some channels may have to be closed after some time. After settling the balances on the blockchain, a new channel could be opened with new liquidity.

8.3 Discussion of applications

The Lightning Network design is only based on the Script language. It is probably one of the best examples to show how the Bitcoin protocol can be extended and improved, simply from the possibilities provided by the Script language. Due to the limitations that the decentralized nature of Bitcoin brings, it may simply in the close future not be possible to achieve the same level of transactions that VISA has (around 1000 transactions/sec). But by constructing transactions in a clever way, and with the possibilities of Script, a new layer such as Lightning Network was suddenly created. It completely relies on the Bitcoin network, but allows for scaling even far beyond what VISA currently needs. Paul Buchheit made a good analogy comparing Bitcoin to the tcp/ip protocol of the Internet, for which all kinds of new protocols can be developed.

But the Lightning Network needs new opcodes, and updates to consensus rules, to work properly. Those are currently being introduced to Bitcoin and will be discussed in the next chapter. The Lightning Network is only the tip of the iceberg of what may become possible with those small changes.

There is another idea called *Sidechains*, which uses Script to create what is called a *2-way peg*. The idea would make it possible to send bitcoins to "other blockchains". Another blockchain could be used for potential different purposes, maybe it uses a different scripting language, or maybe it is the beta version of Bitcoin, which implements new features but may not be as stable. The rules can be whatever those running that sidechain want them to be. Then bitcoins could be send to some specially formed address, which is designed so that it is now out of control of the sender, and anybody else. They are completely immobilized and can only be unlocked if somebody can prove they are no longer being used elsewhere, in particular another blockchain.

Once this immobilisation transaction is sufficiently confirmed, you send a message to the other blockchain, the one you want to use. This message contains a proof that the coins were sent to that special address on the Bitcoin network, that they are therefore immobilized, and crucially, that you were the one who did it. The sidechain creates the exact same number of tokens on its own network and gives you control of them.

The logic above is symmetric. So, at any point, whoever is holding these coins on the sidechain can send them back to the Bitcoin blockchain by creating a special transaction on the sidechain that immobilises the coins on the sidechain. They'll disappear from the sidechain and become available again in Bitcoin, under the control of whoever last owned them on the sidechain.

Such a sidechain project is among other being developed by a company called *RSK Labs*, which wants to extend the Script language of Bitcoin with a turing-complete language. This would allow to move some amount of bitcoins to the sidechain should a contract be needed where a turing-complete language is necessary. After the contract terminates, those coins could then be directly swapped back to “normal” bitcoins.

Bitcoin will do to banks what email did to the postal industry.

Rick Falkvinge

9
What does the future hold?

Changes to the Bitcoin protocol, and in particularly to Script, would be trivial to do with a hard fork. As discussed in Chapter 5, a hard fork applies new rules to the protocol after which it could become irreconcilably divided. The "old" version and "new" version of Bitcoin could then emerge as distinct projects thereafter. But some changes can be rolled out as soft forks, which guarantees that the nodes running the old version oblivious about the changes will still accept all the validations of the nodes running the new version. A soft fork is possible if the new rules allow a subset of the previous rules.

In the following, a change to the Bitcoin protocol called Segregated Witness will be presented, which was proposed in December 2015 by the developers Pieter Wuille et al. [26]. It is an important change, which among other allows the implementation of some applications discussed in Chapter 8. The changes regard the underlying structure of transactions and are very extensive. But as will be shown, Segregated Witness can nevertheless be rolled out with a soft fork, and easily be used in our model of Bitcoin transactions.

9.1 Restructuring: Segregated Witness

Segregated Witness involves a restructuring of the transaction data, such that they become immalleable, and at the same time adds the ability to change the Script language in any desired way, without requiring a hard fork.

As previously discussed, the transaction hash is calculated by using the whole data structure of the transaction, including the scriptSig part that typically contains signatures. But since a signature cannot sign itself, it is of course excluded from the data it signs. Simply by regenerating a signature, the transaction id would change even though the transaction is not functionally different. This makes it difficult to rely on chains of unconfirmed transactions, which as discussed in Section 8.2 is required for the Lightning Network.

Just for clarification, it should be stressed out that as soon as a transaction is included with a block to the Blockchain, it can of course not change its id anymore, since the blockchain is an append-only ledger. Therefore references to transactions that are confirmed in the blockchain are of course reliable!

Segregated Witness solves the problem of immalleability, by moving the scriptSig "out" of the transaction. The "new" transactions using this change must have all their scriptSig fields in its inputs set to empty. To distinguish between "old" and "new" transaction structures, the "new" transactions should use a new version number. Attached to the new transaction structure is then the so called segregated witness data, which is not part of the transaction hash and consists of nothing more than the scriptSig data.

But since now the scriptSig is empty, also the scriptPubKey must change in order for old nodes oblivious about Segregated Witness to accept the new transactions as valid. Nodes in the Bitcoin network using Segregated Witness will only forward them to other new nodes, while old nodes will receive the transaction structure without the segregated witness part. For those old nodes the new transactions will look like Anyone-Can-Spend transactions.

The scriptPubKey for this new type of transaction can be one of two types. The first is like a Pay-To-PubKey-Hash, but instead of actually being a Script program, only the 20 byte HASH160 value of a public key is pushed inside the scriptPubKey. A segregated witness then simply provides the signature and public key, which hashed must yield the hash value from the scriptPubKey. Other than the push operations, there is in other words no Script opcodes directly involved in this transaction type. The transaction type is simply recognized by its pattern, and consensus rules dictate how to treat the provided values, which will simulate the same execution as for Pay-To-PubKey-Hash programs.

The other type is like a Pay-To-Script-Hash. The scriptPubKey consists of the 32 byte SHA256 value of a Script program. The segregated witness to spend such an output will then provide input values followed by a serialized script whose hash matches the value from the scriptPubKey. Just as with P2SH, the serialized script is deserialized and evaluated for the input values and must return true.

There is another detail worth mentioning. Not only is there a hash value inside the scriptPubKey. Also a 1 byte version is prepended to the hash value. The rules defined above corresponds to version 1. This is represented in the scriptPubKey by a call of `OP_0` before the actual push of the hash value. Furthermore, a rule of Segregated Witness is that if a node encounters a version byte it does not know any rules to, it interprets the output as an Any-One-Can-Spend.

In a similar way to how the rules could be changed to allow for Segregated Witness, new versions of Segregated Witness can be easily rolled out as soft forks using the version bytes, and allow for arbitrary changes to the Script language. In fact a new Script language would be defined with every new version of Segregated Witness, but which in the first version is very much like the current one. This is a huge improvement to the Bitcoin protocol, in which as mentioned in Section 6.2, updates to Script can currently only be done for beforehand reserved opcodes, and are of limited behaviour.

But the Bitcoin community is not yet in complete agreement regarding some of the details of Segregated Witness. Nevertheless a lot of applications are already developed relying on Segregated Witness, and it is a very concrete proposal which may probably become a reality within the very near-term future.

As the underlying behaviour does not change with Segregated Witness, our formal model could easily be used after those changes would be applied. The construction holding the scriptSig would instead just be the segregated witness. Two small rules should simply be added to allow for the new specific programs that Segregated Witness uses.

9.2 Academic recognition

Bitcoin is starting to receive more and more academic attention. The University of Pittsburgh, in partnership with MIT, announced in September 2015 the first cryptocurrency scholarly journal called Ledger, which will be peer reviewed and open access.

It will feature full-length papers not only on Bitcoin, but also written about studies of cryptocurrency in general and blockchain technology. It can bring together multiple disciplines from computer science, economics, sociology, physics, law and political science, to discuss new ideas and research.

There are a lot of spaces online to discuss new and ongoing issues in cryptocurrencies, but they can sometimes lack the accountability of an open system of peer review. The issue of the first volume of Ledger is awaited with great anticipation!

Appendices

A
List of opcodes

This appendix shows some rules to opcodes

$$\frac{}{\widetilde{v},\ OP\_NOP.P\ \rightarrow\ \widetilde{v},\ P}\ \text{OP\_NOP : No operation / skip}$$

$$\frac{}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_NIP.P\ \rightarrow\ \widetilde{v} :: v_n,\ P}\ \text{OP\_NIP : Removes the second-to-top stack item.}$$

OP_OVER: Copies the second-to-top stack item to the top.

$$\frac{}{\widetilde{v} :: v_{n-1} :: v_n,\ OP\_OVER.P\ \rightarrow\ \widetilde{v} :: v_{n-1} :: v_n :: v_{n-1},\ P}\ \text{OP\_OVER}$$

$$\frac{i \in I, \widetilde{v} = v_1 :: v_2 :: \cdots :: v_n}{\widetilde{v} :: i,\ OP\_PICK.P\ \rightarrow\ \widetilde{v} :: v_i,\ P}\ \text{OP\_PICK : The item i back in the stack is copied to the top.}$$

OP_ ROLL: The item i back in the stack is moved to the top.

$$\frac{\widetilde{v} = v_1 :: \cdots :: v_n, \widetilde{v}' = v_1 :: \cdots :: v_{i-1} :: v_{i+1} :: \cdots :: v_n}{\widetilde{v} :: i,\ OP\_ROLL.P\ \rightarrow\ \widetilde{v}' :: v_i,\ P}\ \text{OP\_ROLL}$$

OP_ROT: The top three items on the stack are rotated to the left.

$$\frac{}{\widetilde{v} :: v_1 :: v_2 :: v_3,\ OP\_ROT.P \ \rightarrow\ \widetilde{v} :: v_2 :: v_3 :: v_1,\ P}\ \text{OP\_ROT}$$

OP_ SWAP: The top two items on the stack are swapped.

$$\frac{}{\widetilde{v} :: v_1 :: v_2,\ OP\_SWAP.P \ \rightarrow\ \widetilde{v} :: v_2 :: v_1,\ P}\ \text{OP\_SWAP}$$

OP_TUCK: The item at the top of the stack is copied and inserted before the second-to-top item.

$$\frac{}{\widetilde{v} :: v_1 :: v_2,\ OP\_TUCK.P \ \rightarrow\ \widetilde{v} :: v_2 :: v_1 :: v_2,\ P}\ \text{OP\_TUCK} :$$

$$\frac{}{\widetilde{v} :: v_1 :: v_2,\ OP\_2DROP.P \ \rightarrow\ \widetilde{v},\ P}\ \text{OP\_2DROP} :$$ Removes the top two stack items.

$$\frac{\widetilde{v}_{12} = v_1 :: v_2}{\widetilde{v} :: \widetilde{v}_{12},\ OP\_2DUP.P \ \rightarrow\ \widetilde{v} :: \widetilde{v}_{12} :: \widetilde{v}_{12},\ P}\ \text{OP\_2DUP} :$$ Duplicates the top two stack items.

OP_3DUP: Duplicates the top three stack items.

$$\frac{\widetilde{v}_{123} = v_1 :: v_2 :: v_3}{\widetilde{v} :: \widetilde{v}_{123},\ OP\_3DUP.P \ \rightarrow\ \widetilde{v} :: v\widetilde{v}_{123} :: \widetilde{v}_{123},\ P}\ \text{OP\_3DUP}$$

OP_2OVER: Copies the pair of items two spaces back in the stack to the front.

$$\frac{}{\widetilde{v} :: v_1 :: \cdots :: v_4,\ OP\_2OVER.P \ \rightarrow\ \widetilde{v} :: v_1 :: \cdots :: v_4 :: v_1 :: v_2,\ P}\ \text{OP\_2OVER} :$$

OP_2ROT: The top six items on the stack are rotated two to the left.

$$\frac{}{\widetilde{v} :: v_1 :: \cdots :: v_6,\ OP\_2ROT.P \ \rightarrow\ \widetilde{v} :: v_3 :: \cdots :: v_6 :: v_1 :: v_2,\ P}\ \text{OP\_2ROT} :$$

OP_ 2SWAP: Swaps the top two pairs of items.

$$\frac{}{\widetilde{v} :: v_1 :: v_2 :: v_3 :: v_4,\ OP\_2SWAP.P \ \rightarrow\ \widetilde{v} :: v_3 :: v_4 :: v_1 :: v_2,\ P}\ \text{OP\_2SWAP}$$

OP_EQUALVERIFY: Same as OP EQUAL, but runs OP VERIFY afterwards.

$$\frac{}{\widetilde{v},\ OP\_EQUALVERIFY.P\ \rightarrow\ \widetilde{v},\ OP\_EQUAL.OP\_VERIFY.P}\ \text{OP\_EQUALVERIFY}:$$

$$\frac{}{\widetilde{v} :: v_n,\ OP\_1ADD.P\ \rightarrow\ \widetilde{v} :: v_n + 1,\ P}\ \text{OP\_1ADD}:$$ 1 is added to the input.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_NEGATE.P\ \rightarrow\ \widetilde{v} :: -v_n,\ P}\ \text{OP\_NEGATE}:$$ The sign of the input is flipped.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_ABS.P\ \rightarrow\ \widetilde{v} :: |v_n|,\ P}\ \text{OP\_ABS}:$$ The input is made positive.

OP_NOT: If the input is 0 or 1, it is flipped. Otherwise the output will be 0.

$$\frac{v_n = 0}{\widetilde{v} :: v_n,\ OP\_NOT.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_NOT\_1}$$

$$\frac{v_n \neq 0}{\widetilde{v} :: v_n,\ OP\_NOT.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_NOT\_0}$$

$$\frac{v_n = 0}{\widetilde{v} :: v_n,\ OP\_0NOTEQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_0NOTEQUAL\_0}$$

$$\frac{v_n \neq 0}{\widetilde{v} :: v_n,\ OP\_0NOTEQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_0NOTEQUAL\_1}$$

In the following it is assumed that $a \downarrow n_1$ and $b \downarrow n_2$ and $x \downarrow n_3$ holds true.

$$\frac{a + b \downarrow c}{\widetilde{v} :: a :: b,\ OP\_ADD.P\ \rightarrow\ \widetilde{v} :: c,\ P}\ \text{OP\_ADD}:$$ a is added to b.

$$\frac{a - b \downarrow c}{\widetilde{v} :: a :: b,\ OP\_SUB.P\ \rightarrow\ \widetilde{v} :: c,\ P}\ \text{OP\_SUB}:$$ b is subtracted from a.

$$\frac{a \neq 0 \wedge b \neq 0}{\widetilde{v} :: a :: b,\ OP\_BOOLAND.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_BOOLAND}$$

$$\frac{a = 0 \lor b = 0}{\widetilde{v} :: a :: b,\ OP\_BOOLAND.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_BOOLAND}$$

OP_ BOOLOR: If a or b is not 0, the output is 1. Otherwise 0.

$$\frac{a \neq 0 \lor b \neq 0}{\widetilde{v} :: a :: b,\ OP\_BOOLOR.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_BOOLOR1}$$

$$\frac{a = 0 \land b = 0}{\widetilde{v} :: a :: b,\ OP\_BOOLOR.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_BOOLOR0}$$

OP_ NUMEQUAL: Returns 1 if the numbers are equal, 0 otherwise.

$$\frac{a = b}{\widetilde{v} :: a :: b,\ OP\_NUMEQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_NUMEQUAL1}$$

$$\frac{a \neq b}{\widetilde{v} :: a :: b,\ OP\_NUMEQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_NUMEQUAL0}$$

OP_NUMNOTEQUAL: Returns 1 if the numbers are not equal, 0 otherwise.

$$\frac{a \neq b}{\widetilde{v} :: a :: b,\ OP\_NUMNOTEQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_NUMNOTEQUAL\_1}$$

$$\frac{a = b}{\widetilde{v} :: a :: b,\ OP\_NUMNOTEQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_NUMNOTEQUAL\_0}$$

OP_ LESSTHAN_ 1: Returns 1 if a is less than b, 0 otherwise.

$$\frac{a < b}{\widetilde{v} :: a :: b,\ OP\_LESSTHAN.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_LESSTHAN\_1}$$

$$\frac{a \geq b}{\widetilde{v} :: a :: b,\ OP\_LESSTHAN.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_LESSTHAN\_0}$$

OP_GREATERTHAN: Returns 1 if a is greater than b, 0 otherwise.

$$\frac{a > b}{\widetilde{v} :: a :: b,\ OP\_GREATERTHAN.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_GREATERTHAN\_1}$$

$$\frac{a \leq b}{\widetilde{v} :: a :: b,\ OP\_GREATERTHAN.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_GREATERTHAN\_0}$$

OP_LESSTHANOREQUAL: Returns 1 if a is less than or equal to b, 0 otherwise.

$$\frac{a \leq b}{\widetilde{v} :: a :: b,\ OP\_LESSTHANOREQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_LESSTHANOREQUAL\_1}$$

$$\frac{a \nleq b}{\widetilde{v} :: a :: b,\ OP\_LESSTHANOREQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_LESSTHANOREQUAL\_0}$$

OP_GREATERHANOREQUAL: Returns 1 if a is greater than or equal to b, 0 otherwise.

$$\frac{a \geq b}{\widetilde{v} :: a :: b,\ OP\_GREATERTHANOREQUAL.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_GREATERTHANOREQUAL\_1}$$

$$\frac{a \ngeq b}{\widetilde{v} :: a :: b,\ OP\_GREATERTHANOREQUAL.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_GREATERTHANOREQUAL\_0}$$

$$\frac{a < b}{\widetilde{v} :: a :: b,\ OP\_MIN.P\ \rightarrow\ \widetilde{v} :: a,\ P}\ \text{OP\_MIN\_0} : \text{Returns the smaller of a and b.}$$

$$\frac{a \geq b}{\widetilde{v} :: a :: b,\ OP\_MIN.P\ \rightarrow\ \widetilde{v} :: b,\ P}\ \text{OP\_MIN\_1} : \text{Returns the smaller of a and b.}$$

$$\frac{a > b}{\widetilde{v} :: a :: b,\ OP\_MAX.P\ \rightarrow\ \widetilde{v} :: a,\ P}\ \text{OP\_MAX\_0} : \text{Returns the larger of a and b.}$$

$$\frac{a \leq b}{\widetilde{v} :: a :: b,\ OP\_MAX.P\ \rightarrow\ \widetilde{v} :: b,\ P}\ \text{OP\_MAX\_1} : \text{Returns the larger of a and b.}$$

OP_WITHIN: Returns 1 if x is within the specified range (left-inclusive), 0 otherwise.

$$\frac{a \leq x \wedge x < b}{\widetilde{v} :: x :: a :: b,\ OP\_WITHIN.P\ \rightarrow\ \widetilde{v} :: 1,\ P}\ \text{OP\_WITHIN\_1}$$

$$\frac{a > x \vee x \geq b}{\widetilde{v} :: x :: a :: b,\ OP\_WITHIN.P\ \rightarrow\ \widetilde{v} :: 0,\ P}\ \text{OP\_WITHIN\_0}$$

OP_ RIPEMD160:The input is hashed using RIPEMD-160.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_RIPEMD160.P\ \rightarrow\ \widetilde{v} :: ripemd160(v_n),\ P}\ \text{OP\_RIPEMD160}$$

OP_ SHA1: The input is hashed using SHA1.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_SHA1.P\ \rightarrow\ \widetilde{v} :: sha1(v_n),\ P}\ \mathrm{OP\_SHA1}$$

OP_ SHA256: The input is hashed using SHA256.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_SHA256.P\ \rightarrow\ \widetilde{v} :: sha256(v_n),\ P}\ \mathrm{OP\_SHA256}$$

OP_HASH160: The input is hashed twice: first with SHA-256 and then with RIPEMD-160.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_HASH160.P\ \rightarrow\ \widetilde{v} :: sha256(ripemd160(v_n)),\ P}\ \mathrm{OP\_HASH160}$$

OP_HASH256: The input is hashed two times with SHA-256.

$$\frac{}{\widetilde{v} :: v_n,\ OP\_HASH256.P,\ \rightarrow\ \widetilde{v} :: sha256(sha256(v_n)),\ P,}\ \mathrm{OP\_HASH256}$$

References

[1] Gavin Andresen. Bip16: Pay to script hash. http://web.archive.org/web/20160207225440/https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki. Accessed: 2016-02-07.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

[3] Andreas M. Antonopoulos. *Mastering Bitcoin, Unlocking Digital Cryptocurrencies*. O'Reilly Media, 2014.

[4] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[5] Martin J. Bailey. The welfare cost of inflationary finance. *Journal of Political Economy*, 64 (2):93–110, 1956.

[6] BtcDrak, Mark Friedenbach, and Eric Lombrozo. Bip112: Checksequenceverify. http://web.archive.org/web/20160521104127/https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki. Accessed: 2016-05-21.

[7] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *Proceedings on Advances in cryptology*, pages 319–327. Springer-Verlag New York, Inc., 1990.

[8] Andrea Corbellini. Elliptic curve cryptography: a gentle introduction. https://web.archive.org/web/20150521005117/http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/? Accessed: 2015-05-21.

[9] Wei Dai. b-money, an anonymous, distributed electronic cash system. http://web.archive.org/web/20151105064709/http://www.weidai.com/bmoney.txt. Accessed: 2015-11-05.

[10] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO'92*, pages 139–147. Springer, 1992.

[11] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[12] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

[13] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Assccktion for Computing Machinery*, 32 (2):374–382, 1985.

[14] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. Bip68: Relative lock-time using consensus-enforced sequence numbers. https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki. Accessed: 2016-04-18.

[15] Stuart Haber and W Scott Stornetta. Secure names for bit-strings. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 28–35. ACM, 1997.

[16] Brian Hartley and Trevor O Hawkes. *Rings, modules and linear algebra: a further course in algebra describing the structure of Abelian groups and canonical forms of matrices through the study of rings and modules.* Chapman & Hall/CRC, 1970.

[17] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson, 2006.

[18] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[19] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies.* Princeton University Press, 2016.

[20] Rafael Pass. Lecture 21: Collision-resistant hash functions and general digital signature scheme. https://www.cs.cornell.edu/courses/cs6830/2009fa/scribes/lecture21.pdf, 2009.

[21] Nick Szabo. Bit gold. http://web.archive.org/web/20150626152817/http://unenumerated.blogspot.com/2005/12/bit-gold.html. Accessed: 2015-06-26.

[22] Peter Todd. Bip65: Checklocktimeverify. http://web.archive.org/web/20160224213536/https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki. Accessed: 2016-02-24.

[23] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35, Aarhus, Denmark, 2005. Springer.

[24] Xiaoyun Wang, Yiqun L. Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36, Santa Barbara, USA, 2005. Springer.

[25] Pieter Wuille. Bip62: Dealing with malleability. `http://web.archive.org/web/20160404165202/https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`. Accessed: 2016-04-04.

[26] Pieter Wuille, Johnson Lau, and Eric Lombrozo. Bip112: Segregated witness. `http://web.archive.org/web/20160521104121/https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki`. Accessed: 2016-05-21.