(Ab)using Bitcoin for anti-censorship tool

Okupski, K.S.

Award date:
2014
(Ab)using Bitcoin for an Anti-Censorship Tool

KRZYSZTOF OKUPSKI

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
EINDHOVEN UNIVERSITY OF TECHNOLOGY

GRADUATION SUPERVISOR:  
Dr. Boris Škorić

ASSESSMENT COMMITTEE:  
Dr. Boris Škorić  
Dr. Benne de Weger  
Dr. Alexander Serebrenik
## Contents

1 Introduction .................................................. 1  
1.1 Motivation .................................................. 1  
1.2 Project Goals ............................................. 2  
1.3 Outline .................................................... 2  

2 Previous Work .................................................. 4  
2.1 Basic Concepts ............................................. 4  
2.2 Methods .................................................... 4  
2.3 Shortcomings ............................................... 5  

3 Bitcoin Protocol Specification .................................. 7  
3.1 Preliminaries ............................................... 7  
3.1.1 Proof of Work ......................................... 7  
3.1.2 Merkle Trees .......................................... 8  
3.2 Architecture ................................................ 10  
3.2.1 Blocks ................................................ 10  
3.2.2 Transactions .......................................... 13  
3.2.3 Transaction Standardness ............................ 19  
3.3 Bitcoin ownership .......................................... 21  
3.3.1 General ............................................... 21  
3.3.2 Script ................................................ 22  
3.3.3 Standard Transaction Types ......................... 23  
3.3.4 Bitcoin Addresses ................................... 31  
3.3.5 Signatures ............................................ 34  
3.4 Blockchain .................................................. 40  
3.4.1 Structure ............................................. 40  
3.4.2 Mining ................................................. 41  

4 Analysis ......................................................... 43  
4.1 User Control Classification .................................. 43  
4.1.1 Evaluation Framework ................................ 43  
4.1.2 Results ................................................ 43  
4.2 Degrees of Variability ...................................... 45  
4.2.1 Script Components ...................................... 45  
4.2.2 Script Types .......................................... 50  
4.2.3 Transaction Amount ................................... 55  
4.2.4 Lock Time and Sequence Numbers ................... 57  
4.3 Mapping Functions .......................................... 59  
4.3.1 Mapping of Strings to EC Points ..................... 59  
4.3.2 Mapping of Integers to k-Compositions ............... 60  
4.3.3 Mapping of Integers to Permutations ................. 63  
4.4 Experiments ................................................ 65
List of Figures

3.1 Merkle Tree .......................................................... 8
3.2 Merkle Tree with Missing Nodes .................................. 9
3.3 Block Reference Computation ..................................... 11
3.4 Transaction Output Reference Computation ...................... 14
3.5 Bitcoin Transaction Chain ........................................ 21
3.6 Pay-to-Pubkey Structure ......................................... 23
3.7 Pay-to-PubkeyHash Structure .................................... 24
3.8 Pay-to-ScriptHash Structure ...................................... 26
3.9 P2SH Pay-to-PublicKey Structure ................................. 26
3.10 Multisig Structure .................................................. 28
3.11 Nulldata Structure .................................................. 30
3.12 P2PKH Address Computation .................................... 32
3.13 P2SH Address Computation ....................................... 33
3.14 Signature Computation - Initial State ......................... 34
3.15 Signature Computation - SIGHASH_ALL ....................... 35
3.16 Signature Computation - SIGHASH_SINGLE ................... 36
3.17 Signature Computation - SIGHASH_NONE ...................... 37
3.18 Signature Computation - SIGHASH_ALL|SIGHASH_ANYONECANPAY 38
3.19 Signature Computation - Finalization ......................... 39
3.20 Blockchain ............................................................. 40
4.1 Time of Signature Generation .................................... 46
4.2 Time of Keypair Pre-Computation ............................... 48
4.3 Space of Keypair Pre-Computation .............................. 48
4.4 Degree of Variability in Transaction Budget Splitting ....... 56
4.5 Degree of Variability in Transaction Budget Claiming ....... 58
4.6 Scope of Data Embedding Model ................................. 69
4.7 Cost Rate Scaling ................................................... 70
4.8 Transaction Fee Scaling .......................................... 71
5.1 System Architecture ................................................. 72
5.2 Transaction Structure with Embedded Data ..................... 74
5.3 Message Publishing Process Flow ................................. 76
5.4 Message Extraction Process Flow ................................. 77
List of Tables

3.1 Block Structure ........................................... 10
3.2 Regular Transaction Structure ................................... 13
3.3 Lock Time Values ........................................... 16
3.4 Coinbase Transaction Structure ................................... 17
3.5 Coinbase Field Encoding ................................... 18
3.6 Pay-to-Pubkey scriptSig Execution ................................ 23
3.7 Pay-to-Pubkey scriptPubkey Execution ................................ 23
3.8 Pay-to-PubkeyHash scriptSig Execution ................................ 25
3.9 Pay-to-PubkeyHash scriptPubkey Execution ................................ 25
3.10 Pay-to-ScriptHash scriptSig Execution ................................ 27
3.11 Multisig scriptSig Execution ................................ 29
3.12 Multisig scriptPubkey Execution ................................ 29
4.1 Regular Transaction - User Control Classification ......................... 44
4.2 DER-Encoded Signature Structure ................................ 45
4.3 EC Public Key Structure ........................................... 47
4.4 Pay-to-Pubkey Degree of Variability Evaluation ......................... 52
4.5 Pay-to-PubkeyHash Degree of Variability Evaluation .................... 52
4.6 Multisig Degree of Variability Evaluation .......................... 53
4.7 NullData Degree of Variability Evaluation .......................... 53
4.8 P2SH Degree of Variability Evaluation .......................... 54
(Ab)using Bitcoin as an Anti-Censorship Tool
1 Introduction

Bitcoin is a decentralized digital cryptocurrency created by the pseudonymous developer Satoshi Nakamoto. Its origins date back to November 2008, when the first paper on Bitcoin was published [1]. It provides an abstract description of the concepts and architecture of the Bitcoin protocol. It was used as theoretical groundwork for the first implementation of a fully functional Bitcoin client [2], which appeared in January 2009. Ever since it has been a constant source of inspiration for both research and commerce. Its inner workings have been studied extensively and a broad community has emerged, part of which devoted itself to continuously maintain and improve Bitcoin. Finally, over the past years an extensive economy has evolved as well, consisting among others of currency exchange platforms, countless vendors and even a small industry specialized in mining hardware.

1.1 Motivation

Freedom of speech belongs to one of the most fundamental human rights in existence. As such, it has been immortalized in some of the most important human rights documents, dating back as early as the late 17th century with the Bill of Rights. One of the most recent definitions of freedom of speech can be found in the Universal Declaration of Human Rights [3], adopted in 1948 by all member states of the United Nations:

> Everyone has the right to freedom of opinion and expression; this right includes freedom to hold opinions without interference and to seek, receive and impart information and ideas through any media and regardless of frontiers.

It is self-evident that freedom of speech should hold for everyone, yet a significant part of the world’s population is deprived of this basic right. The most notorious example among different countries violating this human right is the People’s Republic of China. It is said that the internet censorship apparatus of the People’s Republic of China belongs to the most extensive and advanced ones in the world and is used to neutralize any opinions deemed unwanted by the state.

Internet censorship in the People’s Republic of China is enforced by both legal and technical means. The former includes laws under which individuals may be fined or imprisoned. These extend to companies as well, where the company can be held accountable for the misbehaviour of their employees and of their customers. For example, Internet Service Providers as well as Media Service Providers are legally responsible for their customer’s conduct. It is therefore common that internet connections as well as chat rooms, forums and other media platforms are supervised internally. The technical means include commonly known filtering methods such as IP blocking, DNS poisoning, URL keyword filtering and limited packet content filtering. Connections utilizing secure transport protocols, such as SSL VPN, IPsec and the like, that could be used to evade censorship, are blocked. In this context we believe that Bitcoin may be utilized to function as an anti-censorship tool.

The first argument for the use of Bitcoin as anti-censorship tool is of political and economical nature. Bitcoin is the first virtual currency of its kind and as such poses
difficulties for legislators to both formulate and enforce laws that fit into the existing legal framework. Furthermore, it possesses significant economical value and it would be disadvantageous to the economy of a country to ban it.

The second argument is of technical nature. Bitcoin possesses the characteristics of being both pseudonymous and decentralized. The former allows a user to act under a random, and if desired temporary, pseudonym rather than a permanent and personally identifiable one. In addition, it is computationally intractable to subvert its authenticity. In conjunction with the loosely-organized and decentralized network structure, the possibilities of enforcing any restrictions upon Bitcoin are limited. It is for these reasons that governments, such as the one of People’s Republic of China, are currently struggling heavily to control its ever increasing growth in popularity.

1.2 Project Goals

The main goals of this thesis can be divided into a theoretical and a practical part. Firstly, the theoretical part focuses on:

i) Describing the inner workings of Bitcoin
ii) Identifying elements that may be used to embed data
iii) Constructing and optimizing a model for data embedding

Considering there is no formal specification of the Bitcoin protocol, it is essential for the purpose of this work that one is created and thoroughly reviewed by the community to ensure correctness and completeness. Next, from the specification elements that may embed data will be identified and evaluated. Finally, a data embedding model will be constructed and its parameters optimized in order to minimize embedding costs.

Secondly, based upon the findings and evaluations in the theoretical part, a proof of concept will be built implemented. The implementation should:

i) Be an extension of the standard Bitcoin software
ii) Publish messages in the blockchain under a pseudonym
iii) Read messages published in the blockchain under a pseudonym

The purpose of the first requirement is to achieve architectural modularity and thereby allow the implementation to focus solely on the messaging functionality. The two latter requirements represent the core functionality of the system, i.e. writing messages into and reading messages from the blockchain.

1.3 Outline

The thesis is divided into five sections. It begins with section 2, consisting of a preliminary literature study. Existing work on using the blockchain for data embedding, including software implementations, is discussed and evaluated. The presented results, in particular the shortcomings, are then taken into account for the development of an improved system.
In section 3, a formal description of the Bitcoin protocol is presented. It provides a
detailed technical view on the inner workings of Bitcoin and is used throughout the re-
mainder of the thesis as a reference. The description covers the most important aspects
of Bitcoin’s architecture, namely the structure and semantics of blocks and transactions.

In section 4, an analysis of various possibilities for data embedding is conducted. First,
elements of regular transactions suitable for data embedding are identified and then the
amount of data that can be embedded in them is determined. Complementary mapping
functions are constructed to embed data into complex data structures. Furthermore, a
sample compression scheme is presented and evaluated to illustrate the possibility to
minimize the overall data size. Finally, a data embedding model is discussed and vari-
able parameters are optimized.

In section 5, the results of the analysis are put into practice and the software design of
the proof of concept is discussed. The architecture and the workflow of the system are
explained in detail.

Section 6 concludes the thesis by summarising it, mentioning encountered issues, de-
scribing the contributions and providing suggestions for future work.
2 Previous Work

Including messages into the blockchain is by no means a novel idea and has been done since the inception of Bitcoin. The very first block ever mined, commonly referred to as the genesis block, contains a political statement on the instability of the banking system, presumably left by the pseudonymous author of Bitcoin - Satoshi Nakamoto [4]:

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

The blockchain, however, contains much more peculiar examples than the one given above. As outlined in [5], these comprise among others random photographs, various documents, such as the Cablegate files, and a python script for including arbitrary data into the blockchain. In the following the various methods for publishing data in the blockchain, as listed in [5], will be discussed.

2.1 Basic Concepts

Prior to discussing the various methods for including data into the blockchain, it is essential to understand the basic concepts of Bitcoin and hence a short introduction will be given. More details can be found in Sect. 3.2 and 3.3.2.

The blockchain is a public ledger which stores all processed transactions in chronological order. Transactions are bundled into a block and processed by a network of so-called miners, which attempt to solve a cryptographic puzzle. Among the unprocessed regular transactions the miner includes a coinbase transaction that transfers a reward to himself for his efforts. The most significant difference between regular and coinbase transactions is that the former can be created by any user and will always be included in a block whereas the latter is created by miners and will only be included in a block if it correctly solves the cryptographic puzzle.

Each transaction consists of a set of transaction inputs and transaction outputs. Transaction inputs and outputs are of a specific type and exist in pairs linked across two individual transactions. Whereas a transaction output essentially specifies an amount of coins and conditions under which these can be claimed, a transaction input references to an output and proves that the specified conditions are met. It is important to note that transaction outputs define in its conditions a public key or a derived form of it, referred to as an address, to identify recipients. Transaction inputs, on the other hand, define a signature to prove that they are in fact the intended recipient.

2.2 Methods

The various known techniques make use of the fact the each Bitcoin user maintains a copy of the blockchain on their local system. These embed data in transactions which are stored in blocks and then included in the into the blockchain. The various approaches can be classified by the transaction type they utilize to embed data with. Generally, since regular transactions can be created by any user and will always be included into
blocks, they are preferred over coinbase transactions.

The most common approach employs regular transactions with transaction outputs of Pay-to-PubkeyHash type (see Sect. 3.3.3). For each transaction output, the address is chosen by the sender and can thus be used to encode arbitrary data. The order of transaction outputs is controlled by the sender as well, so that the order is preserved when a message is split across them. However, since neither the public key corresponding to the address, nor the private key corresponding to the public key are known, the transferred amount is invariably lost.

The second approach extends the previous technique by utilizing transaction outputs of Multisig type (see Sect. 3.3.3). For each transaction output, up to three public keys are chosen by the sender and can therefore be used to encode arbitrary data. Each public key contains approximately threefold the amount of data compared to an address. In consequence, the second approach increases the amount of data stored per transaction output ninefold compared to the first approach. This reduces the relative amount of transaction overhead and thus the costs. Once again, however, the private keys corresponding to the chosen public keys will be unknown and the transferred coins will be lost.

2.3 Shortcomings

The previously discussed methods possess significant shortcomings. These can be attributed to several factors, most notably the simplicity and age of their design. The Bitcoin protocol has changed over time in various ways, on the one hand introducing new freedoms for embedding data and, on the other, restricting existing ones. The shortcomings can be summarized as follows:

1) **Unspendable outputs**
   Since each transaction output has to specify a certain minimum of Bitcoins and the private keys corresponding to the self-chosen public keys or addresses are unknown, the transacted amount is lost. This can be mitigated by utilizing a Multisig transaction type, as presented in the second approach, with a public key owned by the sender.

2) **Outdated transaction input-output types**
   The transaction input-output pairs are of an old type and newer ones exist that allow greater flexibility. In particular, a new transaction type named P2SH Multisig (see Sect. 3.3.3) has been introduced, allowing currently up to six uncompressed or twelve compressed public keys to be used per transaction output. These constraints will be relaxed even further with the next major release of the Bitcoin software.

3) **No efficient data encoding**
   The data included in transactions is typically encoded in ASCII format. Data compression can be performed prior to inclusion to minimize the overall message size.
Learning from these imperfections, in the scope of this thesis an improved approach for embedding data into the blockchain will be presented.
3 Bitcoin Protocol Specification

3.1 Preliminaries

This section gives a short introduction to cryptographic constructs necessary for a thorough understanding of the protocol. In particular, the proof of work scheme and Merkle trees will be discussed. Note that digital signatures are required as well but are intentionally skipped for they are sufficiently covered by online literature.

3.1.1 Proof of Work

A proof of work is a cryptographic puzzle used to ensure that a party has performed a certain amount of work. In particular, the Bitcoin mining process (see Sect. 3.4.2) incorporates a proof of work system based on Adam Back’s Hashcash [6]. It has two basic properties - firstly, it ensures that the party providing the proof of work has invested a predefined amount of effort in order to create the proof and secondly, that the proof is efficiently verifiable. Typically, finding a solution to a proof of work puzzle is a probabilistic process with a success probability depending on the predefined difficulty.

Let Alice and Bob be two parties communicating with each other and let Alice require Bob to perform a certain amount of computational work for each message he sends to Alice. To do so, Alice can require Bob to provide a string whose one-way hash satisfies a predefined structure. Finding such a string has a certain success probability that will determine how much work Bob has to invest on average in order to find a valid solution.

For example, in Bitcoin the hashing algorithm is \( \text{double-SHA256} \) (\( \text{SHA256}^2 \)) and the predefined structure is a hash less or equal to a target value \( T \). The success probability of finding a nonce \( n \) for a given message \( msg \), such that \( H = \text{SHA256}^2(msg||n) \) is less or equal to the target \( T \) is

\[
Pr[H \leq T] = \frac{T}{2^{256}}
\]  

This will require a party attempting to find a proof of work to perform, on average, the following amount of computations

\[
\frac{1}{Pr[H \leq T]} = \frac{2^{256}}{T}
\]

Finally, it is easy to see that it can be efficiently verified whether the nonce accompanied with the message is indeed a valid proof of work by simply evaluating

\[
\text{SHA256}^2(msg||n) \leq T
\]
3.1.2 Merkle Trees

Merkle trees, named after their creator Ralph Merkle, are binary hash trees used for efficient verification of data integrity. An example of a Merkle tree can be seen in Fig. 3.1. Leaves are computed directly as hashes over data blocks, whereas nodes further up the tree are computed by concatenating and hashing their respective children.

![Merkle Tree Diagram]

The main advantage of Merkle trees is that when one data block changes it is not necessary to compute a hash over all the data, as opposed to naive hashing. Assume data block $d_{00}$ is modified, then $n_{00}$ has to be re-computed as well as all nodes along the branch until the root node. Therefore, the number of required hash computations scales logarithmically in the number of data blocks. Since both data blocks and hashes are relatively small in size, this process is fairly efficient.

**Other cases**

The previously discussed example considered the situation where the number of data blocks is a power of two. In such a case the computation results in a full and complete binary tree. However, since it is required that each node, except for the leaves, has exactly two children, measures have to be taken if nodes are missing. In the following the method used in Bitcoin will be discussed.

The solution is straightforward - when forming a row in the tree (excluding the root), whenever there is an odd number of nodes, the last node is duplicated. In effect, each intermediary row in the tree will always have an even number of nodes and therefore each node, except for the leaves, will have exactly two children.
In the example given in Fig. 3.2 there are only three data blocks and therefore the computation of the fourth node in the second last row is missing a child. Thus, the last node is replicated and the computation is continued as in the previous example (see Fig. 3.1). Should an odd number of nodes occur at any other point during the computation, then the same rule is applied.
3.2 Architecture

Central to Bitcoin’s architecture is a public ledger called the **blockchain**, which stores all processed **transactions** in chronological order. Transactions are processed by a loosely-organized network of **miners** in a process called **mining** (see Sect. 3.4.2). In it the miner creates a **block** with a set of unprocessed transactions and attempts to solve a **proof of work puzzle** (see Sect. 3.1.1). Once a valid solution has been found, the block including the solution is published throughout the network and accepted into the blockchain. In this section the structure of blocks and transactions will be discussed in detail. Note that the following description is based on the Bitcoin source code [2] and the Bitcoin Protocol Specification on Wikipedia [7]. Furthermore, all data types denoted in the diagrams are explained in detail in Appendix A.

### 3.2.1 Blocks

Each block is composed of a header and a payload. The header stores the current block header version (**nVersion**), a reference to the previous block (**HashPrevBlock**), the root node of the Merkle tree (**HashMerkleRoot**), a timestamp (**nTime**), a target value (**nBits**) and a nonce (**nNonce**). Finally, the payload stores the number of transactions (**#vtx**) and the vector of transactions (**vtx**) included in the block.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type (Size)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVersion</td>
<td>int (4 bytes)</td>
<td>Block format version (currently 2).</td>
</tr>
<tr>
<td>HashPrevBlock</td>
<td>uint256 (32 bytes)</td>
<td>Hash of previous block header SHA256²(nVersion</td>
</tr>
<tr>
<td>HashMerkleRoot</td>
<td>uint256 (32 bytes)</td>
<td>Top hash of the Merkle tree built from all transactions.</td>
</tr>
<tr>
<td>nTime</td>
<td>unsigned int (4 bytes)</td>
<td>Timestamp in UNIX-format of approximate block creation time.</td>
</tr>
<tr>
<td>nBits</td>
<td>unsigned int (4 bytes)</td>
<td>Target T for the proof of work problem in compact format. Full target value is derived as: $T = 0xh_2h_3h_4h_5h_6h_7 * 2^{256-2(x_0h_0h_1-3)}$</td>
</tr>
<tr>
<td>nNonce</td>
<td>unsigned int (4 bytes)</td>
<td>Nonce allowing variations for solving the proof of work problem.</td>
</tr>
<tr>
<td>#vtx</td>
<td>VarInt (1-9 bytes)</td>
<td>Number of transaction entries in vtx.</td>
</tr>
<tr>
<td>vtx[]</td>
<td>Transaction (Variable)</td>
<td>Vector of transactions.</td>
</tr>
</tbody>
</table>

*Table 3.1: Block Structure*
nVersion
The version field stores the version number of the block format. Ever since BIP0034 [8] is in place, the block format version is 2 and blocks of any other version are neither relayed nor mined.

HashPrevBlock
This field stores the reference to the previous block, computed as a hash over the block header as depicted in Fig. 3.3.

\[
SHA256^2(nVersion || HashPrevBlock || HashMerkleRoot || nTime || nBits || nNonce)
\]

The reference functions as a chaining link in the blockchain. By including a reference to the previous block, a chronological order on blocks, and thus transactions as well, is imposed.

HashMerkleRoot
This field stores the root of the Merkle hash tree. It is used to provide integrity of all transactions included in the block and is computed according to the scheme described in Sect. 3.1.2. The parameters used for computing the tree are double-SHA256 as the hashing algorithm and raw transactions as the data blocks (see Table 3.2 and 3.4).

nTime
The time field stores the timestamp in UNIX format denoting the approximate block creation time. As the timestamp is a parameter included in block mining, it is fixed at the beginning of the process.

nBits
The nBits field stores a compact representation of a target value $T$, which is utilized in the proof of work puzzle (see Sect. 3.4.2). The target value is a 256 bit long number, whereas its corresponding compact representation is only 32 bits long and thus encoded
with only 8 hex digits. The target value can be derived from its compact hexadecimal representation $0xh_0h_1h_2h_3h_4h_5h_6h_7$ with the formula

$$0xh_2h_3h_4h_5h_6h_7 \ast 2^{8 \ast (0xh_0h_1 - 3)}$$ (5)

The upper bound for the target is defined as $0x1D00FFFF$ whereas there is no lower bound. The very first block, the genesis block, has been mined using the maximum target. In order to ensure that blocks are mined at a constant rate of one block per 10 minutes throughout the network, the target $T$ is recalculated every 2016 blocks. This is done based on the time $t_{sum}$ it took to mine, due to an off-by-one error [9], the last 2015 blocks:

$$T' = \frac{t_{sum}}{14 \ast 24 \ast 60 \ast 60s} \ast T$$ (6)

Note that $t_{sum}$ is calculated as the difference of the timestamps $nTime$ in the block header.

**nNonce**

The nonce field contains arbitrary data and is used as a source of randomness for solving the proof of work problem. However, since it is fairly small in size with 4 bytes, it does not necessarily provide sufficient variation for finding a solution. Therefore, other sources exist and will be addressed in more detail in Sect. 3.4.2.
3.2.2 Transactions

In principle, there are two types of transactions, coinbase transactions and regular transactions. Coinbase transactions are special transactions in which new Bitcoins are introduced into the system. They are included in every block as the very first transaction and are meant as a reward for solving a proof of work puzzle. Regular transactions, on the other hand, are used to transfer existing Bitcoins amongst different users. From an architectural point of view, a coinbase transaction can be seen as a special case of a regular transaction. For this reason, the structure of a regular transaction will be discussed first, followed by the differences between coinbase and regular transactions.

Regular transactions

As mentioned in the previous section, each block in the blockchain includes a set of transactions. Every transaction consists of a transaction version (nVersion), a vector of inputs (vin) and a vector of outputs (vout), both preceded by their count, and a transaction inclusion date (nLockTime).

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type (Size)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVersion</td>
<td>int (4 bytes)</td>
<td>Transaction format version (currently 1).</td>
</tr>
<tr>
<td>#vin</td>
<td>VarInt (1-9 bytes)</td>
<td>Number of transaction input entries in vin.</td>
</tr>
<tr>
<td>hash</td>
<td>uint256 (32 bytes)</td>
<td>Double-SHA256 hash of a past transaction.</td>
</tr>
<tr>
<td>n</td>
<td>uint (4 bytes)</td>
<td>Index of a transaction output within the transaction specified by hash.</td>
</tr>
<tr>
<td>scriptSigLen</td>
<td>VarInt (1-9 bytes)</td>
<td>Length of scriptSig field in bytes.</td>
</tr>
<tr>
<td>scriptSig</td>
<td>CScript (Variable)</td>
<td>Script to satisfy spending condition of the transaction output (hash,n).</td>
</tr>
<tr>
<td>nSequence</td>
<td>uint (4 bytes)</td>
<td>Transaction input sequence number.</td>
</tr>
<tr>
<td>#vout</td>
<td>VarInt (1-9 bytes)</td>
<td>Number of transaction output entries in vout.</td>
</tr>
<tr>
<td>nValue</td>
<td>int64_t (8 bytes)</td>
<td>Amount of $10^{-8}$ BTC.</td>
</tr>
<tr>
<td>scriptPubkeyLen</td>
<td>VarInt (1-9 bytes)</td>
<td>Length of scriptPubkey field in bytes.</td>
</tr>
<tr>
<td>scriptPubkey</td>
<td>CScript (Variable)</td>
<td>Script specifying conditions under which the transaction output can be claimed.</td>
</tr>
<tr>
<td>nLockTime</td>
<td>unsigned int (4 bytes)</td>
<td>Timestamp past which transactions can be replaced before inclusion in block.</td>
</tr>
</tbody>
</table>

Table 3.2: Regular Transaction Structure

nVersion

The version field stores the version number of the transaction format. The current transaction format version is 1.
# vin
This field stores the number of elements in the inputs vector vin. It is encoded as a variable length integer (see Appendix A).

vin
The vin field stores a vector of one or more transaction inputs. Each transaction input is composed of a reference to a previous output (hash,n), the length of the signature script field in bytes (scriptSigLen), the signature script field (scriptSig) itself and a transaction sequence number (nSequence).

- (hash,n)
A previous output is uniquely identified by the tuple (hash,n). The hash field, also referred to as the transaction ID (TxID), is computed as a double-SHA256 hash of the raw transaction:

$$TxID = SHA256^2(Tx)$$ (7)

Hence, whilst a transaction is uniquely identified by its hash, the specific output within that transaction is identified by the output index n. An example is given below in Fig. 3.4.

![Figure 3.4: Transaction Output Reference Computation](image)
- **scriptSigLen**
  This field stores the length of the signature script field `scriptSig` in bytes. It is encoded as a variable length integer (see Appendix A).

- **scriptSig**
  The signature script field contains a response script corresponding to the challenge script (see `scriptPubkey` field) of the referenced transaction output (`hash`, `n`). More precisely, whilst the challenge script specifies conditions under which the transaction output can be claimed, the response script is used to prove that the transaction is allowed to claim it. More details on transaction verification can be found in Sect. 3.3.2.

- **nSequence**
  This field stores the transaction input sequence number. It was once intended for multiple signers to agree to update a transaction before including it in a block. If a signer was done updating, he marked his transaction input as final by setting the sequence number to the highest 4-byte integer value `0xFFFFFFFF`. More details can be found in Sect. 3.2.3 under the Final Transaction Rule.

### vout
This field stores the number of elements in the output vector `vout`. It is encoded as a variable length integer (see Appendix A).

**vout**
The `vout` field stores a vector of one or more transaction outputs. Each transaction output is composed of an amount of BTC to be spent (`nValue`), the length of the public key script (`scriptPubkeyLen`) and the public key script (`scriptPubkey`) itself.

- **nValue**
  The `nValue` field stores the amount of BTC to be spent by the output. The amount is encoded in Satoshis, that is $10^{-8}$ BTC, allowing tiny fractions of a Bitcoin to be spent. However, note that in the reference implementation transactions with outputs less than a certain value are referred to as “dust” and are considered non-standard [10]. This value is currently by default 546 Satoshi and can be defined by each node manually. Dust transactions are neither relayed nor mined. More details on dust transactions can be found in Appendix B.

- **scriptPubkeyLen**
  This field stores the length of the public key script `scriptPubkey` in bytes. It is encoded as a variable length integer (see Appendix A).

- **scriptPubkey**
  The public key script field contains a challenge script for transaction verification. More precisely, whilst the challenge script specifies conditions under which the transaction output can be claimed, the response script (see `scriptSig` field) is used to prove that the transaction is allowed to claim it. More details on transaction verification can be found in Sect. 3.3.2.

**nLockTime**
This field stores the lock time of a transaction, i.e. a point in time past which the
transaction should be included in a block. Once the lock time has been exceeded, the transaction is locked and can be included in a block. The lock time is encoded as either a timestamp in UNIX format or as a block number:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Always locked.</td>
</tr>
<tr>
<td>(&lt;5 \times 10^8)</td>
<td>Block number at which transaction is locked.</td>
</tr>
<tr>
<td>(\geq5 \times 10^8)</td>
<td>UNIX timestamp at which transaction is locked.</td>
</tr>
</tbody>
</table>

\textbf{Table 3.3: Lock Time Values}

If all transaction inputs (see \textit{vin} field) have a final sequence number (see \textit{nSequence} field), then the lock time is ignored. More details can be found in Sect. 3.2.3 under the Final Transaction Rule.
Coinbase Transactions

As can be seen in Table 3.4, except for renaming the signature script field from *scriptSig* to *coinbase*, the data structure of the transaction remains the same. However, there are several constraints specific to a coinbase transaction. In the following the differences between a regular and a coinbase transaction will be explained.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type (Size)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVersion</td>
<td>int (4 bytes)</td>
<td>Transaction format version (currently 1).</td>
</tr>
<tr>
<td>#vin</td>
<td>VarInt (1-9 bytes)</td>
<td>Number of transaction inputs entries in <em>vin</em>.</td>
</tr>
<tr>
<td>vin[]</td>
<td>hash uint256 (32 bytes)</td>
<td>Fixed double-SHA256 hash.</td>
</tr>
<tr>
<td></td>
<td>n uint (4 bytes)</td>
<td>Fixed transaction output index.</td>
</tr>
<tr>
<td></td>
<td>coinbaseLen VarInt (1-9 bytes)</td>
<td>Length of <em>coinbase</em> field in bytes.</td>
</tr>
<tr>
<td></td>
<td>coinbase CScript (Variable)</td>
<td>Encodes the block height and arbitrary data.</td>
</tr>
<tr>
<td></td>
<td>nSequence uint (4 bytes)</td>
<td>Transaction input sequence number.</td>
</tr>
<tr>
<td>#vout</td>
<td>VarInt (1-9 bytes)</td>
<td>Number of transaction output entries in <em>vout</em>.</td>
</tr>
<tr>
<td>vout[]</td>
<td>nValue int64 (8 bytes)</td>
<td>Amount of $10^{-8}$ BTC.</td>
</tr>
<tr>
<td></td>
<td>scriptPubkeyLen VarInt (1-9 bytes)</td>
<td>Length of <em>scriptPubkey</em> field in bytes.</td>
</tr>
<tr>
<td></td>
<td>scriptPubkey CScript (Variable)</td>
<td>Script specifying conditions under which the transaction output can be claimed.</td>
</tr>
<tr>
<td></td>
<td>nLockTime unsigned int (4 bytes)</td>
<td>Timestamp until which transactions can be replaced before block inclusion.</td>
</tr>
</tbody>
</table>

Table 3.4: Coinbase Transaction Structure

#vin

The number of inputs stored in the input vector *vin* is always 1.

vin

The *vin* field stores a vector of precisely one transaction input. The input is composed of a fixed transaction output reference (*hash*, *n*), the length of the coinbase field in bytes (*coinbaseLen*), the coinbase field (*coinbase*) itself and a transaction sequence number (*nSequence*).

- (*hash*, *n*)

In a coinbase transaction new coins are introduced into the system and therefore no previous transaction output is referenced. The (*hash*, *n*) tuple stores the following constant values:

$$hash = 0$$
$$n = 2^{32} - 1$$  \(8\)
- coinbaseLen
  This field stores the length of the coinbase field coinbase in bytes. It is in the range of 2-100 bytes and is encoded as a variable length integer (see Appendix A).

- coinbase
  The coinbase field, also referred to as the coinbase script, stores the block height, i.e. the block number within the blockchain, and arbitrary data.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockHeightLen</td>
<td>1</td>
<td>Length of blockHeight field in bytes.</td>
</tr>
<tr>
<td>blockHeight</td>
<td>blockHeightLen</td>
<td>Block height encoding.</td>
</tr>
<tr>
<td>arbitraryData</td>
<td>coinbaseLen – (blockHeightLen+1)</td>
<td>Arbitrary data field.</td>
</tr>
</tbody>
</table>

Table 3.5: Coinbase Field Encoding

As of BIP0034 [8], the beginning of the coinbase field is reserved for the block height. It is encoded in serialized Script format, i.e. the first byte specifies the size of the block height in bytes, followed by the block height itself in little-endian notation. The remaining bytes can be chosen arbitrarily and provide variability for the proof of work puzzle (see Sect. 3.4.2).

vout
The transaction output vector is constrained by the maximal amount of Bitcoins that is allowed to be transacted. More precisely, there are certain rules that dictate how the nValue field is supposed to be calculated.

- nValue
  In a coinbase transaction the miner is allowed to claim the current mining subsidy, as well as transaction fees for all included transactions, as a reward for solving the proof of work puzzle. The subsidy for finding a valid block is currently 25 BTC and is halved every 210000 blocks. The transaction fee, on the other hand, is computed for each transaction as the difference between the sum of input values and the sum of output values.
3.2.3 Transaction Standardness

Transaction standardness is defined as a set of requirements that is enforced upon a transaction by any node utilizing the reference client for transaction processing. Transactions that do not meet all the requirements are considered non-standard and will be neither relayed nor mined. Note that these rules are not enforced upon transactions of an already mined block. It is thus allowed to mine and include non-standard transactions in blocks. The transaction standardness rules are as follows.

**Transaction Size**
A single transaction may not exceed 10000 bytes in size.

**Transaction Version**
The transaction format version is currently 1.

**Final Transaction Rule**
A transaction is called final if it satisfies at least one of the following conditions:

1) The transaction lock time (see \(nLockTime\) field) is set to locked or has been exceeded.

2) All transaction inputs are final (see \(nSequence\) field).

This rule is associated with an obsolete mechanism called transaction replacement. It allowed to replace certain parts of a transaction, e.g. transaction inputs, until either all transaction inputs were finalized or the transaction lock time had passed. Note, however, that the transaction replacement functionality has been completely removed from the reference implementation to reduce the complexity of the protocol. Moreover, although the transaction lock time functionality is still in place, it is considered non-standard.

**Transaction Input Rules**
For each transaction input the following requirements must be satisfied by each signature script field:

1) *Signature Script Size*
The size may not exceed 500 bytes. Note that this limitation will change to 1650 bytes in the next major release of the reference client.

2) *Push Only*
Only a restricted set of data push operations is allowed. To be specific, only opcodes\(^1\) in the range 0x00-0x60 are permitted.

3) *Canonical Pushes*
The scripting language allows to push data on the stack in different ways. This rule enforces that only data pushes intended for a particular data size are allowed.

\(^1\) See [https://en.bitcoin.it/wiki/Script](https://en.bitcoin.it/wiki/Script) for complete reference.
Transaction Output Rules
For each transaction output the following requirements must be satisfied:

1) *Standard Transaction Type*
   The public key script field (*scriptPubkey*) must encode a standard transaction type (see Sect. 3.3.3).

2) *Non-Dust Transaction*
   The transaction is not allowed to be “dust”. A transaction is called “dust” if it contains an output that spends more than one third in transaction fees (see Appendix B for calculation of the fee).

Nulldata Transaction Count
At most one transaction output of Nulldata transaction type (see Sect. 3.3.3) per transaction is permitted.
3.3 Bitcoin ownership

3.3.1 General

Bitcoin utilizes two cryptographic primitives to realize a secure and decentralized transaction authorization system. Firstly, it employs asymmetric cryptography for (i) identification and (ii) authentication of recipients, as well as to (iii) ensure integrity of regular transactions. More precisely, the public key of a public/private keypair is used to identify a particular recipient, whereas the private key is used to create a signature for both transaction authentication and integrity.

Secondly, the proof of work protocol is used to (i) regulate coin supply, (ii) reward miners for transaction processing and (iii) ensure block integrity. In the mining process all regular transactions of users and a special coinbase transaction created by the miner are processed by solving the proof of work problem. It is important to note that while the authenticity and integrity of regular transactions is ensured by the previously discussed signature scheme, the integrity of coinbase transactions is assured by the proof of work puzzle. In the following the exact application of these primitives will be described with the help of Fig. 3.5.

![Bitcoin Transaction Chain](image)

To begin with, Bitcoins are introduced into the system with coinbase transactions (see Sect. 3.2.2). In it the miner specifies one or more transaction outputs (vout), defining the amounts and destinations to which the freshly created coins are to be transferred. He identifies each destination by including a public key or a derived form of it in the public key script field (scriptPubkey). As discussed above, the integrity of the coinbase transaction is ensured by the computational hardness of the proof of work problem.

Next, when the miner intends to spend his reward, he creates a regular transaction, references it to the specific output of the coinbase transaction and provides a signature in the signature script field (scriptSig). Since the signature is computed over the complete transactions (see Sect. 3.3.5), control of the private key corresponding to the referenced
public key is proven and integrity of the transaction is guaranteed. This chain is continued indefinitely and logged publicly in the blockchain to keep track of all Bitcoins within the system at all times.

3.3.2 Script

Script is a stack-based, Turing-incomplete language designed specifically for the Bitcoin protocol. A script is essentially a set of instructions\(^2\) that are processed left to right. Script is used to encode two components - a challenge script and a response script:

- A challenge script (see scriptPubkey field) is part of a transaction output and specifies under which conditions it can be claimed.
- A response script (see scriptSig field) is part of a transaction input and is used to prove that the referenced transaction output can be rightfully claimed.

For a given transaction, each transaction input is verified by first evaluating scriptSig, then copying the resulting stack and finally evaluating scriptPubkey of the referenced transaction output. If during the evaluation no failure is triggered and the final top stack element yields true, then the ownership has been successfully verified.

Although Script is very comprehensive and allows one to construct intricate conditions under which coins can be claimed, much of its functionality is currently disabled in the reference implementation and only a restricted set of standard script templates is accepted. These are Pay-to-Pubkey (P2PK), Pay-to-PubkeyHash (P2PKH), Pay-to-ScriptHash (P2SH), Multisig and Nulldata. In the next section, the structure of these standard transaction types will be discussed.

3.3.3 Standard Transaction Types

Pay-to-Pubkey (P2PK)

The structure of the challenge and response scripts of a Pay-to-Pubkey transaction are depicted below in Fig. 3.6. Note that operations in a script are written as OP_X, where OP stands for operation and X is an abbreviation of the operation’s function. For example, in Fig. 3.6 CHECKSIG stands for signature verification.

```
| ScriptPubkey: <pubkey> OP_CHECKSIG |
| scriptSig:    <signature>          |
```

**Figure 3.6: Pay-to-Pubkey Structure**

In a Pay-to-Pubkey transaction the sender transfers Bitcoins directly to the owner of a public key. He specifies in the challenge script the public key (pubkey) and one requirement that the claimant has to prove:

1) Knowledge of the private key corresponding to the public key.

To do so, the claimant creates a response script containing only a signature. The scripts are then evaluated as depicted in Table 3.6 and 3.7. The signature and the public key are pushed onto the stack and evaluated. Note that the signature is computed as described in Sect. 3.3.5.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>&lt;signature&gt;</td>
<td>The signature is pushed on the stack.</td>
</tr>
<tr>
<td>&lt;signature&gt;</td>
<td>Empty</td>
<td>Final state after evaluating scriptSig.</td>
</tr>
</tbody>
</table>

**Table 3.6: Pay-to-Pubkey scriptSig Execution**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;signature&gt;</td>
<td>&lt;pubkey&gt; OP_CHECKSIG</td>
<td>State after copying the stack of the signature script evaluation. The public key is pushed on the stack.</td>
</tr>
<tr>
<td>&lt;pubkey&gt; &lt;signature&gt;</td>
<td>OP_CHECKSIG</td>
<td>The signature is verified for the top two stack elements and the result is pushed on the stack.</td>
</tr>
<tr>
<td>True</td>
<td>Empty</td>
<td>Final state after evaluating scriptPubkey.</td>
</tr>
</tbody>
</table>

**Table 3.7: Pay-to-Pubkey scriptPubkey Execution**
Pay-to-PubkeyHash (P2PKH)
The structure of the challenge and response scripts of a Pay-to-PubkeyHash transaction can be seen below in Fig. 3.7.

| scriptPubkey: | OP_DUP OP_HASH160 <pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG |
| scriptSig: | <signature> <pubkey> |

Figure 3.7: Pay-to-PubkeyHash Structure

In a Pay-to-PubkeyHash transaction the sender transfers Bitcoins to the owner of a P2PKH address (see Sect. 3.3.4). He specifies in the challenge script the public key hash (pubkeyHash) of the Bitcoin address (depicted in Fig. 3.12) and two requirements that the claimant has to prove:

1) Knowledge of the public key corresponding to pubkeyHash.
2) Knowledge of the private key corresponding to the public key.

To do so, the claimant creates a response script containing a signature and a public key. The scripts are then evaluated as depicted in Table 3.8 and 3.9. First, it is verified if the public key (pubkey) corresponds to the public key hash (pubkeyHash) and then whether the signature is valid. The signature is computed as described in Sect. 3.3.5.
<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>&lt;signature&gt; &lt;pubkey&gt;</td>
<td>The signature and the public key are pushed on the stack.</td>
</tr>
<tr>
<td>&lt;pubkey&gt; &lt;signature&gt;</td>
<td>Empty</td>
<td>Final state after evaluating scriptSig.</td>
</tr>
</tbody>
</table>

Table 3.8: Pay-to-PubkeyHash scriptSig Execution

<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;pubkey&gt; &lt;signature&gt;</td>
<td>OP_DUP OP_HASH160 &lt;pubkeyHash&gt; OP_EQUALVERIFY OP_CHECKSIG</td>
<td>State after copying the stack of the signature script evaluation. The top stack element is duplicated.</td>
</tr>
<tr>
<td>&lt;pubkey&gt; &lt;pubkey&gt; &lt;signature&gt;</td>
<td>OP_HASH160 &lt;pubkeyHash&gt; OP_EQUALVERIFY OP_CHECKSIG</td>
<td>The top stack element is first hashed with SHA256 and then with RIPEMD160.</td>
</tr>
<tr>
<td>&lt;pubkeyHashNew&gt; &lt;pubkey&gt; &lt;signature&gt;</td>
<td>&lt;pubkeyHash&gt; OP_EQUALVERIFY OP_CHECKSIG</td>
<td>The public key hash is pushed on the stack.</td>
</tr>
<tr>
<td>&lt;pubkeyHash&gt; &lt;pubkeyHashNew&gt; &lt;pubkey&gt; &lt;signature&gt;</td>
<td>OP_EQUALVERIFY OP_CHECKSIG</td>
<td>Equality of the top two stack elements is checked. If it evaluates to true then execution is continued. Otherwise it fails.</td>
</tr>
<tr>
<td>&lt;pubkey&gt; &lt;signature&gt;</td>
<td>OP_CHECKSIG</td>
<td>The signature is verified for the top two stack elements.</td>
</tr>
<tr>
<td>True</td>
<td>Empty</td>
<td>Final state after evaluating scriptPubkey.</td>
</tr>
</tbody>
</table>

Table 3.9: Pay-to-PubkeyHash scriptPubkey Execution
Pay-to-ScriptHash (P2SH)
The structure of the challenge and response scripts of a Pay-to-ScriptHash transaction is depicted below in Fig. 3.8.

\[
\text{scriptPubkey: OP_HASH160 } \langle \text{scriptHash} \rangle \text{ OP_EQUAL} \\
\text{scriptSig: } \langle \text{signatures} \rangle \{\text{serializedScript}\}
\]

**Figure 3.8: Pay-to-ScriptHash Structure**

In a Pay-to-ScriptHash transaction the sender transfers Bitcoins to the owner of a P2SH Bitcoin address (see Sect. 3.3.4). He specifies in the challenge script the serialized script hash (\textit{scriptHash}) of the Bitcoin address (depicted in Fig. 3.13) and one requirement that the claimant has to prove:

1) Knowledge of the redemption script \textit{serializedScript} corresponding to \textit{scriptHash}.

To do so, the claimant creates a response script containing one or more signatures and the serialized redemption script \textit{serializedScript}. Note that unlike in any other standard transaction type the responsibility of supplying the conditions for redeeming the transaction is shifted from the sender to the redeemer. The redeemer may specify any conditions in the redemption script \textit{serializedScript} conforming to standard transaction types. For example, he may define a standard Pay-to-Pubkey transaction as a Pay-to-ScriptHash transaction as follows:

\[
\text{scriptPubkey: OP_HASH160 } \langle \text{scriptHash} \rangle \text{ OP_EQUAL} \\
\text{scriptSig: } \langle \text{signatures} \rangle \{\langle \text{pubkey} \rangle \text{ OP_CHECKSIG}\}
\]

**Figure 3.9: P2SH Pay-to-PublicKey Structure**

Due to the nested nature of this transaction type, the script evaluation requires an additional validation step. First, it is verified whether the redemption script (\textit{serializedScript}) is consistent with the redemption script hash (\textit{scriptHash}) and then the transaction is evaluated using the redemption script as \textit{scriptPubkey}. The evaluation is depicted in Table 3.10, 3.11 and 3.12.
The signature and the redemption script are pushed on the stack.

Final state after evaluating scriptSig.

Table 3.10: Pay-to-ScriptHash scriptSig Execution

State after copying the stack of the signature script evaluation. The top stack element is first hashed with SHA256 and then with RIPEMD160.

The redemption script hash is pushed on the stack.

Equality of the top two stack elements is checked. The result of the evaluation is pushed on the stack.

Final state after evaluating scriptPubkey.

Table 3.11: Pay-to-ScriptHash scriptPubkey Execution

For the additional validation step the stack after scriptSig execution (see Table 3.10) is copied, the top stack element is popped and used as the script. The state now resembles the beginning of a standard Pay-to-Pubkey transaction evaluation (see Table 3.7).

Table 3.12: Pay-to-ScriptHash Supplementary Validation
Multisig
The structure of the challenge and response scripts of a Multisig transaction is depicted below in Fig. 3.10.

\[
\text{scriptPubkey: } m \text{ <pubkey 1> \ldots <pubkey n> n OP\_CHECKMULTISIG} \\
\text{scriptSig: } \text{OP\_0 <signature 1> \ldots <signature m>}
\]

Figure 3.10: Multisig Structure

In a Multisig transaction the sender transfers Bitcoins to the owner of \( m \)-of-\( n \) public keys. He specifies in the challenge script \( n \) public keys (\( \text{pubkey 1..n} \)) and a requirement that the claimant has to prove:

1) Knowledge of at least \( m \) private keys corresponding to the public keys.

To do so, the claimant creates a response script containing at least \( m \) signatures in the same order of appearance as the public keys. Note that due to an off-by-one error \text{OP\_CHECKMULTISIG} pops one too many elements off the stack and it is therefore required to prepend the response script with a zero data push \text{OP\_0}. The script is then evaluated as depicted in Table 3.13 and 3.14. First, the signatures are pushed on the stack, followed by the number of required signatures \( m \), the public keys and the number of public keys \( n \).

The bounds for a standard Multisig transaction are \( 1 \leq m \leq n \leq 3 \), whereas for a P2SH Multisig transaction they are variable. The upper bound for a P2SH Multisig transaction is restricted by both the allowed size of the signature script \( \text{scriptSig} \) (500 bytes) and the allowed size of the serialized script \( \text{serializedScript} \) (520 bytes). It is therefore possible to create e.g. a 1-of-12 P2SH Multisig transaction with compressed public keys or a 4-of-5 P2SH Multisig transaction with compressed public keys. Note that the maximum size of the signature script field (\( \text{scriptSig} \)) will be increased in the next major release to 1650 bytes, thus allowing even bigger P2SH Multisig transactions.
The signatures are pushed on the stack.

Final state after evaluating `scriptSig`.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>OP_0 &lt;signature 1&gt; ... &lt;signature m&gt;</td>
<td>The signatures are pushed on the stack.</td>
</tr>
<tr>
<td>&lt;signature m&gt;</td>
<td>...</td>
<td>Empty</td>
</tr>
<tr>
<td>&lt;signature 1&gt;</td>
<td>OP_0</td>
<td>Final state after evaluating <code>scriptSig</code>.</td>
</tr>
</tbody>
</table>

Table 3.13: Multisig `scriptSig` Execution

State after copying the stack of the signature script evaluation. The public keys are pushed on the stack.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Remaining Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;signature m&gt;</td>
<td>...</td>
<td>m &lt;pubkey 1&gt; ... &lt;pubkey n&gt; n OP_CHECKMULTISIG</td>
</tr>
<tr>
<td>&lt;signature 1&gt;</td>
<td>OP_0</td>
<td>State after copying the stack of the signature script evaluation. The public keys are pushed on the stack.</td>
</tr>
<tr>
<td>&lt;pubkey n&gt;</td>
<td>...</td>
<td>OP_CHECKMULTISIG</td>
</tr>
<tr>
<td>&lt;pubkey 1&gt;</td>
<td>m</td>
<td>The signatures are verified in order of appearance and the result is pushed on the stack.</td>
</tr>
<tr>
<td>&lt;signature m&gt;</td>
<td>...</td>
<td>OP_0</td>
</tr>
<tr>
<td>&lt;signature 1&gt;</td>
<td>OP_0</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>Empty</td>
<td>Final state after evaluating <code>scriptPubkey</code>.</td>
</tr>
</tbody>
</table>

Table 3.14: Multisig `scriptPubkey` Execution
Nulldata

The structure of the challenge and response scripts of a Nulldata transaction is depicted below in Fig. 3.11.

```
| scriptPubkey: OP_RETURN [SMALLDATA] |
| scriptSig:                             |
```

**Figure 3.11: Nulldata Structure**

Unlike all other standard transaction types, a Nulldata transaction does not specify in the challenge script any recipients and does not have a corresponding response script. Another characteristic of it is that it does not adhere to the dust transaction rule (see Appendix B) and therefore the transaction output value can be set to zero.

The purpose of Nulldata transactions is to allow inclusion of arbitrary data in transactions in a controlled fashion. For this reason these transactions possess an optional field in which up to 40 bytes of data can be stored. Note, however, that in order to prevent blockchain flooding only one output of this type is permitted in a transaction.
3.3.4 Bitcoin Addresses

A Bitcoin address is a unique, 27-34 alphanumeric characters long identifier that can be used as a destination for Bitcoin payments. There are currently two different types of Bitcoin addresses in existence, Pay-to-PubkeyHash and Pay-to-ScriptHash, which are used in conjunction with their corresponding transaction type. In the following both will be described in detail.

Pay-to-PubkeyHash Address

Essentially, a Pay-to-PubkeyHash address is a hash of the public key portion of the public-private ECDSA keypair with a built-in checksum. Schematics of how it is calculated can be seen in Fig. 3.12.

First, the EC public key is hashed using SHA256 and RIPEMD160. The resulting structure is referred to as _pubkeyHash_. Next, a constant version byte is prepended to _pubkeyHash_. A checksum is built over it by applying a double-SHA256 hash and truncating the result to the first 4 bytes. The checksum is then appended. Finally, the result is converted into a human-readable string using Base58 encoding [11]. The final result is a _P2PKH address_.

Pay-to-ScriptHash Address

A Pay-to-ScriptHash address on the other hand, is a hash of the redemption script _serializedScript_, with a built-in checksum. Schematics of how it is calculated can be seen in Fig. 3.13.

First, the redemption script is hashed using SHA256 and RIPEMD160. The resulting structure is referred to as _scriptHash_. Next, a constant version byte is prepended to _scriptHash_. A checksum is built over it by applying a double-SHA256 hash and truncating the result to the first 4 bytes. The checksum is then appended. Finally, the result is converted into a human-readable string using Base58 encoding [11]. The final result is a _P2SH address_.


Figure 3.12: P2PKH Address Computation

*EC Public Key encoded as an uncompressed point on the secp256k1 curve according to the ANSI X9.62 standard
Figure 3.13: P2SH Address Computation
3.3.5 Signatures

Signatures are a central cryptographic primitive in Bitcoin and play a significant role in transaction authorization (see Sect. 3.3.1). In a regular transaction, a signature is included in the signature script field (scriptSig) of every transaction input to prove that the referenced transaction output can be rightfully spent by the claimant.

Hash Types

Signatures in Bitcoin are of a specific type, referred to as hash type, that determines which parts of the transaction are covered by it. This allows the signer to selectively choose which transaction parts should be protected and which parts can be modified by others. Three base signature hash types available, the default type SIGHASH_ALL, SIGHASH_SINGLE and SIGHASH_NONE. Additionally, a special type modifier called SIGHASH_ANYONECANPAY can be applied in conjunction with one of the three base types.

![Signature Computation - Initial State](image)

**Figure 3.14:** Signature Computation - Initial State

In the following the various signature types will be discussed in detail. The depiction in Fig. 3.14 shows the initial state of a sample transaction which will be used to illustrate the process. The signature will be performed for the first transaction input of the transaction **TxNew**, which references the first output of a past transaction **TxPrev**.
SIGHASH_ALL

The default signature hash type SIGHASH_ALL represents the simplest of the three base types. It signs the complete transaction, including all the transactions inputs and outputs, with the exception of the signature script fields. The coverage of the signature is illustrated below in Fig. 3.15 with grey fields.

Before the signature is computed, several temporary changes are made to the transaction:

a) The signature script of the currently signed input is replaced with the public key script, excluding all occurrences of OP_CODESEPARATOR in it, of the referenced transaction output.

b) The signature scripts of all other inputs are replaced with empty scripts.
**SIGHASH_SINGLE**

In the second signature hash type SIGHASH_SINGLE all transaction inputs and the transaction output corresponding to the currently signed input is signed. The coverage of the signature is illustrated below in Fig. 3.16 with grey fields.

Before the signature is computed, several temporary changes are made to the transaction:

a) The signature script of the currently signed input is replaced with the public key script, excluding all occurrences of `OP_CODESEPARATOR` in it, of the referenced transaction output.

b) For all the remaining transaction inputs:
   - The signature scripts are replaced with empty scripts.
   - The sequence number is set to zero.

c) The number of transaction outputs is set to the currently signed transaction input index plus one.

d) All transaction outputs up to the currently signed one are emptied.

![Signature Computation - SIGHASH_SINGLE Diagram](image_url)
SIGHASH_NONE

In the third signature hash type *SIGHASH_NONE* all transaction inputs and none of the transaction outputs are signed. The coverage of the signature is illustrated below in Fig. 3.17 with grey fields.

**Figure 3.17:** Signature Computation - SIGHASH_NONE

Before the signature is computed, several temporary changes are made to the transaction:

a) The signature script of the currently signed input is replaced with the public key script, excluding all occurrences of `OP_CODESEPARATOR` in it, of the referenced transaction output.

b) For all the remaining transaction inputs:
   - The signature scripts are replaced with empty scripts.
   - The sequence number is set to zero.

c) The number of transaction outputs is set to zero.

d) All transaction outputs are removed.
SIGHASH_ANYONECANPAY

The `SIGHASH_ANYONECANPAY` modifier is used in conjunction with a base type and affects the signature coverage of transaction inputs. It is used to only cover the currently signed input by the signature. For example, the transaction depicted in Fig. 3.18 illustrates that the second transaction input is excluded from the signature.

![Signature Computation - SIGHASH_ALL|SIGHASH_ANYONECANPAY](image)

In addition to the changes performed by the base hash type, the following temporary changes are made before the signature is computed:

a) The number of transaction inputs is set to one.

b) All transaction inputs, except for the currently signed one, are removed.
Finalization
Once the transaction type has been chosen and the hash type dependent modifications have been applied, the actual signature is computed. This is done as follows - first, the hash type is appended to the transaction, then the signature itself is computed and finally the hashtype is appended to it. The ECDSA signature is computed using double-SHA256 and the secp256k1 elliptic curve as parameters. The appended hashtype signals the verifying party what hash type was applied.

Figure 3.19: Signature Computation - Finalization
3.4 Blockchain

3.4.1 Structure

The blockchain is a record of all transactions that have occurred in the Bitcoin system and is shared by every node in it. Its main purpose is to infer a list of all unspent transaction outputs and their spending conditions. The novelty of Bitcoin lies, among other things, in how the blockchain is structured in order to guarantee chronological ordering of transactions and prevent double-spending in a distributed network.

As described in Sect. 3.2.1, every block in the blockchain refers to the hash of a previous block. This imposes a chronological order on blocks and therefore transactions as well, since it is not possible to create a valid hash of the previous block header prior to its existence.

Furthermore, each block includes the solution to a proof of work puzzle of a certain difficulty. The computational power involved in solving the proof of work puzzle for each block is used as a voting scheme to enable all nodes in the network to collectively agree on a version of the blockchain. In particular, nodes agree on the blockchain that involved the highest accumulated computational effort to be created. Thus, modifying a block in the chain would require an adversary to recompute proof of work puzzles of equal or greater computational effort than the ones from that block up to the newest block. In order to achieve this, the adversary would have to computationally outperform the majority of the network, which is considered infeasible.

![Blockchain Diagram](image-url)

**Figure 3.20:** Blockchain

Clearly, since nodes in the network compete in a randomized process to successfully solve the proof of work puzzle and gain a reward, there is a chance that two different blocks are mined simultaneously and the chain forks. In this case nodes will accept whichever block they have received first and continue building the chain upon that block. If another block is found, then the branch that was used will become the main...
blockchain. If this happens, all valid transactions within the shorter chain are re-added to the pool of queued transactions. The resulting structure resembles what is depicted in Fig. 3.20, the white block being the first block ever mined, also referred to as the genesis block, the black chain representing the main chain and grey blocks being orphans due to forking.

3.4.2 Mining

Procedure

The process of finding a valid block is called *mining* whereas nodes that participate in that process are called *miners*. As described in [12], mining nodes perform the following steps in an endless loop:

1) Collect all broadcasted transactions and validate whether they satisfy the miner’s self-defined policy. Typically, a transaction includes a transaction fee that functions as an incentive for the miner to include it in the block. However, if it does not, then it is up to the miner to decide whether or not to include it.

2) Verify all transactions that are to be included in the block. Transactions are verified as described in Sect. 3.3.2 and it is checked whether their inputs have been previously spent.

3) Select the most recent block on the longest path in the blockchain, i.e. the path that involves most accumulated computational effort, and insert the hash of the block header into the new block.

4) Solve the proof of work problem as described below and broadcast the solution. Should another node solve the proof of work problem before, then the block is first validated, meaning the proof of work solution is checked and all transactions included in the block are verified. If it passes these controls then the cycle is repeated. Note that if there are transactions that have not been included in the new block then they are saved and included in the next cycle.

Proof of Work

During mining a miner attempts to find a block header whose double-SHA256 hash lies below the target value $T$. In order to succeed he needs a certain degree of freedom in the block header that allows him to compute various hashes without interfering with its semantics. Hence, two fields are used as a source of randomness - the nonce field ($n\text{Nonce}$) in the block header itself and the coinbase field ($\text{coinbase}$) in the coinbase transaction, which indirectly changes the Merkle root ($\text{HashMerkleRoot}$) in the block header. The process of finding a proof of work can then be divided into three steps:

1) Set the nonce field and the coinbase field to values of one’s choice.

2) Compute the hash of the block header as

$$SHA256^2(n\text{Version}||\text{HashPrevBlock}||\text{HashMerkleRoot}||n\text{Time}||n\text{Bits}||n\text{Nonce})$$

(9)
3) Reverse the byte order of the computed hash and check whether its value $H$ lies below the current target value $T$ (stored in compact format in the $nBits$ field):

$$H \leq T$$

(10)

This process is repeated for various nonce and coinbase values until a valid solution is found. Typically, for efficiency reasons, all possible values of the nonce field are evaluated before changes to the coinbase field are made.
4 Analysis

The goal of this section is to design an optimal model for data inclusion. In order to do so, we will first perform an analysis of various degrees of variability in a transaction, based on (i) the protocol specification and (ii) complementary experiments. Additionally, to further minimize data embedding costs, we will consider a sample data compression technique. Finally, based on the prior analysis, we will construct and evaluate a data embedding model.

4.1 User Control Classification

A regular transaction consists of several fields of simple and complex data types. For some of them only fixed, predetermined values are allowed, whereas for others nearly arbitrary, user-chosen values are permitted. We make a preliminary classification of these fields in order to determine which of them are promising for data inclusion.

4.1.1 Evaluation Framework

Transaction fields may be roughly classified, according to the degree of control a user has over them, in the the following four categories:

1) *Arbitrary*
   
   There are no restrictions enforced upon this field and its value can be chosen arbitrarily within its specified data type.

2) *High*
   
   There are weak restrictions enforced upon this field. For example, the value must lie within certain bounds or be of a predefined structure.

3) *Low*
   
   There are strong restrictions enforced upon this field. Its value can be influenced only in a very limited fashion.

4) *None*
   
   The restrictions enforced upon this field do not allow it to be manipulated in any way. In other words, its value is fixed.

A sensible baseline for this reference is to classify fields that scored high or better as potential candidates for data inclusion. Fields that score lower than the baseline simply do not possess sufficient variance to qualify as a feasible option.

4.1.2 Results

Firstly, there is one field over which there is absolutely no user control, namely the transaction version field. The transaction version is fixed and is only incremented when the transaction structure changes.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Type (Size)</th>
<th>User Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVersion</td>
<td>int (4 bytes)</td>
<td>None</td>
</tr>
<tr>
<td>#vin</td>
<td>VarInt (1-9 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>hash</td>
<td>uint256 (32 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>n</td>
<td>uint (4 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>scriptSigLen</td>
<td>VarInt (1-9 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>scriptSig</td>
<td>CScript (Variable)</td>
<td>High</td>
</tr>
<tr>
<td>nSequence</td>
<td>uint (4 bytes)</td>
<td>High</td>
</tr>
<tr>
<td>#vout</td>
<td>VarInt (1-9 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>nValue</td>
<td>int64_t (8 bytes)</td>
<td>High</td>
</tr>
<tr>
<td>scriptPubkeyLen</td>
<td>VarInt (1-9 bytes)</td>
<td>Low</td>
</tr>
<tr>
<td>scriptPubkey</td>
<td>CScript (Variable)</td>
<td>High</td>
</tr>
<tr>
<td>nLockTime</td>
<td>unsigned int (4 bytes)</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 4.1: Regular Transaction - User Control Classification

Secondly, there are several fields with low user control. These are the transaction input and output counters, script length indicators and the transaction output reference. It is fairly easy to see that in the case of the transaction input and output counters, as well as the script length indicators, the value can only be influenced indirectly. The value of the transaction output reference, on the other hand, can only be chosen from a very restricted set over which there is nearly no control.

Finally, there are several fields with high user control. These are the public key and signature scripts, the transaction output value, the transaction input sequence number and the transaction lock time. The public key and signature scripts, as the following analysis will show, must be of a predefined structure within which certain components can be set nearly arbitrarily. The transaction output value must lie within specific bounds, where the lower and upper bounds are determined by the dust transaction rule (see Appendix B) and the sum of claimed transaction output values, respectively. As to the transaction input sequence numbers and the transaction lock time, it follows from the final transaction rule (see Sect. 3.2.3) that the value of either one of them can be chosen freely if the other is fixed.
4.2 Degrees of Variability

The previous results have shown which fields are potentially eligible for the purpose of arbitrary data inclusion. In this section these fields will be evaluated in more detail in order to attain precise values for the amount of data that can be included. Note that only fields whose score is high or better will be considered.

4.2.1 Script Components

There are currently five standard transaction types, defined in Sect. 3.3.3, that determine the structure of public key scripts and signature scripts. The basic building blocks of scripts are signatures, public keys, hashes, data fields and operations. Depending on the transaction type, they are (i) arranged in a specific way and (ii) occur in different quantities. Hence, we first determine the degree of variability in individual components prior to the analysis of scripts as a whole.

Signatures

Signatures are realized using the ECDSA signature scheme. The ECDSA scheme has two variable parameters [13], an elliptic curve with specific domain settings and an auxiliary hash function. In Bitcoin the Koblitz Curve secp256k1 with domain parameters specified in [14] and double-SHA256 are used, respectively. As described in Appendix B, a signature for a message $m$ is a pair $(R, S)$ computed as

$$R \equiv (K \cdot G)_x \pmod{n}$$
$$S \equiv (h(m) + d \cdot R) \cdot K^{-1} \pmod{n}$$

where $K$ is a random integer, $G$ is a generator of a group with the order $n$, $d$ is the private key and $h$ is a hash function. These are DER (Distinguished Encoding Rules) encoded and their individual elements, including the supplementary signature hash type, can be seen below in Table 4.2.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>1 byte</td>
<td>ASN.1 sequence indicator, fixed to 0x30.</td>
</tr>
<tr>
<td>SigLen</td>
<td>1 byte</td>
<td>Total signature length in bytes.</td>
</tr>
<tr>
<td>R-Type</td>
<td>1 byte</td>
<td>R value type indicator, fixed to 0x02 (integer).</td>
</tr>
<tr>
<td>LenR</td>
<td>1 byte</td>
<td>Length of value R in bytes.</td>
</tr>
<tr>
<td>R</td>
<td>33 bytes</td>
<td>Value R (Padded 32 byte integer).</td>
</tr>
<tr>
<td>S-Type</td>
<td>1 byte</td>
<td>S value type indicator, fixed to 0x02 (integer).</td>
</tr>
<tr>
<td>LenS</td>
<td>1 byte</td>
<td>Length of value S in bytes.</td>
</tr>
<tr>
<td>S</td>
<td>33 bytes</td>
<td>Value S (Padded 32 byte integer).</td>
</tr>
<tr>
<td>HashType</td>
<td>1 byte</td>
<td>Type of applied signature hash type.</td>
</tr>
</tbody>
</table>

Table 4.2: DER-Encoded Signature Structure

It can be deduced from the list of individual signature elements that three elements may be controlled by the user, the component $R$, the component $S$ and the signature hash type $HashType$. In the following each of these possibilities will be discussed in more detail.
R-Component The $R$ component encodes a uniformly random 32-byte integer. Its value can be, in theory, chosen freely and thereby it has a degree of variability of

$$256 \text{ bits}$$

(11)

Although the $R$ component represents a feasible source for data inclusion, there are security risks entailed if it is used in practice. It plays an essential role in the security of the ECDSA signature scheme and its incorrect use can lead to the leak of the private key. In particular, if the space of $R$ is reduced, then the probability of two independently and randomly chosen values of $R$ colliding increases. If there is an instance where for two different signatures the private key $d$ and $R$ are identical, then the private key can be derived efficiently. This is explained in more detail in Appendix D.

S-Component The ECDSA signature generation is a randomized process where each time a signature is computed, a new ephemeral key $K_E$ is chosen that in turn influences the value of $S$. This fact can be used to derive the following scheme.

Let $n$ be a desired degree of variability in bits, then generate the signature repeatedly for randomly chosen values of $K_E$ until the trailing $n$ bits of $S$ correspond to the bits chosen by the user.

As illustrated by Fig. 4.1, this approach possesses an exponential time complexity since it takes on average $2^n$ iterations, each lasting approximately 2ms, until the desired trailing bits of $S$ are obtained.

![Figure 4.1: Time of Signature Generation](image)

Signature Hash Type The signature hash type element is used to inform verifiers which hash type was used to create the signature. As defined in Sect. 3.3.5, there are only three base types, which combined with the auxiliary modifier, results in six distinct types. The hash type can thus take only one out of six predefined values and therefore has a degree of variability of

$$\lfloor \log_2 6 \rfloor = 2 \text{ bits}$$

(12)
In addition to the low degree of variability the hash type changes the coverage of the signature and for that reason it will be dismissed as a feasible source for data inclusion.

Public Keys

A public key is a ANSI X9.62 encoded point \((x, y)\) on the secp256k1 elliptic curve with domain parameters specified in the Standards for Efficient Cryptography [14] and can be stored in either compressed or uncompressed form.

\[
\begin{array}{|c|c|}
\hline
\text{Field Name} & \text{Prefix} & \text{X} \\
\hline
\text{Size (bytes)} & 1 & 32 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|}
\hline
\text{Field Name} & \text{Prefix} & \text{X} & \text{Y} \\
\hline
\text{Size (bytes)} & 1 & 32 & 32 \\
\hline
\end{array}
\]

(a) Compressed  
(b) Uncompressed

Table 4.3: EC Public Key Structure

For comparison, the structure of a compressed and an uncompressed point on the curve are illustrated above in Table 4.3. The first byte indicates whether the point is compressed or not. If it is, the first byte denotes whether the Y-coordinate is even (02\text{16}) or odd (03\text{16}) and is followed by the 32-byte long X-coordinate. Together they are used to recover the Y-coordinate and thus save space. If the point is not compressed, then the first byte (04\text{16}) signifies that point compression is not used and is followed by the 32-byte long X and Y-coordinates.

The amount of data that can be included in a public key depends on whether the private key corresponding to the public key is known or not. In the following both cases will be covered in more detail.

Known Private Key  Provided that the private key corresponding to a public key has to be known, arbitrary data can be encoded into the public key by using the fact that the keypair generation process is randomized.

Let \(n\) be a desired degree of variability in bits, then pre-compute a set of public-private keypairs containing all combinations of the trailing \(n\) bits of the public key. The \(n\)-bit suffix of the public key is then under explicit user control and can be chosen arbitrarily.
Note that the degree of variability $n$ is limited by both the time and space complexity of the approach. During the pre-computation of the table an amount of time and space exponential in $n$ has to be expended. Once the table has been computed, it only needs to be used for look-up operations.

The exponential space complexity is more of a bottleneck than the exponential time complexity since it can only be expended at an additional cost. Each entry in the table consists of a 33 byte compressed public key and a 32 byte private key. Due to their exponential count the table size swiftly increases to infeasible amounts.

**Unknown Private Key** If the private key corresponding to a public key need not be known, then the amount of data that can be included depends on whether the public
key has to be a valid curve point or not.

First we need to clarify whether the tuple \((x, y)\) must represent a valid point on the curve. Experiments conducted in Sect. 4.4.1 show that the use of invalid points is permitted by miners. However, this policy may be changed in the near future to prevent clogging of the blockchain with unnecessary data. Transactions employing invalid public keys can be detected with little effort and marked as non-standard, in which case they would be neither relayed nor mined. Furthermore such transactions are clearly distinguishable from regular transactions and may be suppressed by a surveillance state with simple filtering methods. For this reason we will refrain from using invalid points on the curve and consider the use of valid points only.

In order to determine the amount of data we can include into valid public keys we shall consider the mapping function \(\phi\) described in Sect. 4.3.1. It maps an \(n\)-bit string to a valid compressed point on the curve. The string length is restricted to at most 250 bits which we shall mark as the upper bound for the amount of data that can be included in public keys. Note that through the use of compressed points we achieve a better embedded data density since the uniquely determined \(y\)-coordinate is not stored.

**Hashes**

Hashes represent a poor option for data inclusion for two reasons, (i) the pre-image is longer than the hash and (ii) the is little control over the hash. The pre-image of a hash can be either a public key or a redemption script, both of which are longer than the truncated hash and thereby there is a loss in degree of variability. Furthermore the hash can be influenced only indirectly over the pre-image and therefore there is no reason why the pre-image itself should not be utilized instead of the hash.

**Data Fields**

It is clear that the optional data field in Nulldata transactions can be used to include up to 320 bytes of arbitrary data. However, due to the transaction standardness rules only one transaction output of Nulldata type per transaction is permitted.

**Operations**

Operations have a fixed value and position within transactions of a standard transaction type and since only those are relayed and mined, there is no possibility to include data.
4.2.2 Script Types

Evaluation Framework
In this section an evaluation of the data embedding potential in script pairs of standard transaction types will be performed. The evaluation will take into consideration the following factors:

1) Degree of Variability
   Total amount of data in bits that can be included.

2) Total Script Size
   Overall script size in bytes.

3) Data Density
   The ratio between the degree of variability and total script size.

4) Cost Rate
   Costs of including data in Satoshi per byte.

The cost rate will be calculated at the rate of 10 Satoshi per byte. This follows from the average of the default transaction fee rate that currently lies at 10000 Satoshi per kB (1000 bytes).

Furthermore there are two cases that will be considered during the evaluation for each script pair, one where the corresponding transaction output can be spent and one where it can not. In the former case the private key corresponding to a specified public key has to be known whereas in the latter it does not and we therefore achieve different results.

Note that transaction outputs must specify, according to the dust transaction rule (see Sect. 3.2.3), a minimum amount which in case of unspendable transactions cannot be claimed. For this reason the cost rate in unspendable transactions will be additionally penalized with that amount.

Framework Decisions
The following evaluation will be performed on the basis of the previous analysis on degrees of variability in individual components. However, some of these components possess a flexible degree of variability and total size. The former depends on the amount of time and storage space a user is prepared to expend whereas the latter is an uncontrollable factor that stems from the use of compact encoding schemes for numbers. In order to simplify the evaluation, the degrees of variability and the size of components will be fixed to specific values.

Signatures  We determined that the only feasible degree of variability in signatures lies in the $S$ component. Since this value has to be computed approximately $2^n$ times for each signature, we have decided to set the degree of variability to 8 bits in order to prevent unacceptable delays.

Furthermore, signatures are encoded using the DER encoding scheme which stores both $R$ and $S$ in their most compact form. The signature size fluctuates depending on their values and will therefore be fixed to its largest possible size of 73 bytes.
**Analysis**

**Public Keys**  We distinguish the degree of variability in public keys for two cases, one where the corresponding private key is known and one where it is not. In the first case we shall set the degree of variability to 16 bits whereas in the latter case we set it to 250 bits.

Furthermore we will assume the use of compressed public keys which have a fixed size of 33 bytes.

**Hashes**  Hashes do not possess any degree of variability and has a fixed size of 20 bytes.

**Data Field**  Data fields have a degree of variability of 320 bits and a fixed size of 41 bytes.

**Operations**  Operations do not have any degree of variability and have a fixed size of 1 byte.

Note that the signature, public key and hash components are considered in the context of a script as data and are therefore preceded by a push operation. A complete list of the different push operations and their sizes can be found in [15].
Pay-to-Pubkey

The public key located in the public key script of a transaction output as well as the signature located in the signature script of a transaction input can be used to include arbitrary data.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Degree of Variability (Bits)</th>
<th>Script Size (Bytes)</th>
<th>Data Density</th>
<th>Cost Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PK (spendable)</td>
<td>16</td>
<td>35</td>
<td>2.75%</td>
<td>363.33 S/B</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2PK (unspendable)</td>
<td>250</td>
<td>35</td>
<td>89.29%</td>
<td>29.63 S/B</td>
</tr>
</tbody>
</table>

**Table 4.4:** Pay-to-Pubkey Degree of Variability Evaluation

In the spendable variant there is one public key and one signature that can be used to encode 16 bits and 8 bits of information, respectively.

In the unspendable variant, on the other hand, there is only one public key that can encode 250 bits of information. Note that although the transferred amount cannot be claimed, the achieved cost rate is much lower.

Pay-to-PubkeyHash

The public key and the signature are both located in the signature script of a transaction input and can be used to embed data.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Degree of Variability (Bits)</th>
<th>Script Size (Bytes)</th>
<th>Data Density</th>
<th>Cost Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2PKH (spendable)</td>
<td>0</td>
<td>25</td>
<td>2.26%</td>
<td>443.33 S/B</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2PKH (unspendable)</td>
<td>0</td>
<td>25</td>
<td>0%</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Table 4.5:** Pay-to-PubkeyHash Degree of Variability Evaluation

Since the signature and the public key are both located in the signature script, it is necessary that the corresponding private key is known in order to embed any data. As a consequence, we can embed 8 bits in the signature and 16 bits in the corresponding public key using the spendable variant.
Multisig
There are between one and three public keys located in the public key script of a transaction output as well as between one and three signatures located in the signature script of a transaction input that can be used for data inclusion.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Degree of Variability (Bits)</th>
<th>Script Size (Bytes)</th>
<th>Data Density</th>
<th>Cost Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multisig (spendable)</td>
<td>516</td>
<td>105</td>
<td>36.39%</td>
<td>27.48 S/B</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multisig (unspendable)</td>
<td>750</td>
<td>105</td>
<td>89.29%</td>
<td>19.58 S/B</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.6: Multisig Degree of Variability Evaluation**

Since public keys are generally capable of storing more data per byte than signatures, a sensible approach is to maximize the number of public keys and to minimize the number of signatures. Hence we will consider the case where three public keys and one signature are used.

In order to make the transaction output spendable, it is sufficient that the private key corresponding to one out of three public keys is known. We can therefore embed 8 bits in one signature, 16 bits in the corresponding public key and 500 bits in the two remaining public keys.

In the unspendable variant, on the other hand, we can embed 750 bits of information in three public keys. The results show that once again we achieve a higher data density as well as a lower cost rate by using the unspendable variant as compared to the spendable variant.

Nulldata
The Nulldata transaction type was introduced to allow inclusion of arbitrary data in a controlled fashion. For this reason the transaction output has been made provably unspendable and does not have a corresponding transaction input.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Degree of Variability (Bits)</th>
<th>Script Size (Bytes)</th>
<th>Data Density</th>
<th>Cost Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nulldata</td>
<td>320</td>
<td>42</td>
<td>95.24%</td>
<td>10.25 S/B</td>
</tr>
</tbody>
</table>

**Table 4.7: Nulldata Degree of Variability Evaluation**

The data field is located in the public key script of a transaction output and can store up to 320 bits of arbitrary data. Note, however, that only one transaction output of this type can be used per transaction.
Pay-to-ScriptHash

The P2SH transaction type is used in conjunction with either a P2PK, P2PKH or Multisig transaction type. It embeds the public key script of the nested type in the signature script of a transaction input. In the following all three possible combinations will be evaluated.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Degree of Variability (Bits)</th>
<th>Script Size (Bytes)</th>
<th>Data Density</th>
<th>Cost Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2SH P2PK (spendable)</td>
<td>0</td>
<td>23</td>
<td>2.26%</td>
<td>443.33 S/B</td>
</tr>
<tr>
<td>P2SH P2PK (unspendable)</td>
<td>-</td>
<td>-</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>P2SH P2PKH (spendable)</td>
<td>0</td>
<td>23</td>
<td>1.91%</td>
<td>523.33 S/B</td>
</tr>
<tr>
<td>P2SH P2PKH (unspendable)</td>
<td>-</td>
<td>-</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>P2SH Multisig (spendable)</td>
<td>0</td>
<td>23</td>
<td>67.73%</td>
<td>14.77 S/B</td>
</tr>
<tr>
<td>P2SH Multisig (unspendable)</td>
<td>-</td>
<td>-</td>
<td>0%</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 4.8: P2SH Degree of Variability Evaluation

The components that can be used for data inclusion are signatures and public keys. Depending on the nested transaction type, they occur in different quantities in the signature script of the transaction input. For this reason we cannot include any data into unsendable transactions.

We can see in Table 4.8 that whilst P2SH does not bring any advantages when used in combination with P2PK or P2PKH, it does yield significant advantages when it is used with Multisig. This stems from the relaxed constraints on the count of public keys that may be included. In Multisig transactions the number of public keys is limited to three whereas in P2SH Multisig transactions it is limited to twelve.

Analogously to Multisig, only one private key corresponding to one of the twelve public keys has to be known in order to make the transaction spendable. In conclusion, we can embed 8 bits in the signature, 16 bits in the corresponding public key and 2750 bits in the remaining eleven public keys.
4.2.3 Transaction Amount

The transaction amount specifies the number of Satoshis that is transferred by a transaction output. In the following we will present a method that embeds data in the split of the budget that satisfies the dust transaction constraint.

Budget Splitting

Let \( n \) define the spendable budget, \( k \) the number of transaction outputs, \((x_1, ..., x_k)\) the vector of transaction output values and \((l_1, ..., l_k)\) the corresponding lower bounds.

Then for \( n' = n - \sum_{i=1}^{k} l_i \) there are

\[
\binom{n' + k - 1}{k - 1}
\]

ways in which the budget \( n \) can be split into \( k \) values, such that

\[
x_1 + \cdots + x_k = n, \quad x_i \geq l_i \text{ for } i = 1 \ldots k
\]

Method For a given budget \( n \) and a number of transaction outputs \( k \) we can encode an integer \( d \) with \( 0 \leq d < \binom{n'+k-1}{k-1} \) as a vector of integers \((x_1, ..., x_k)\). This is also referred to as a combinatorial composition of an integer and can be computed as

\[
x_i = \psi(n', k, d)[i] + l_i \text{ for } i = 1 \ldots k
\]

where the function \( \psi \), described in Sect. 4.3.2, computes the \( d \)-th lexicographically ordered composition of the integer \( n' \) into \( k \) parts.

The integer \( d \) can then be retrieved from the vector of integers \((x_1, ..., x_k)\) and the corresponding lower bounds \((l_1, ..., l_k)\), where \( x'_i = x_i - l_i \text{ for } i = 1 \ldots k \), by

\[
d = \binom{n' + k - 1}{k - 1} - 1 - \sum_{i=1}^{k-1} \left( \sum_{j=k-i+1}^{k} x'_j + i - 1 \right)
\]

For more background information see the description provided in Sect. 4.3.2.

Analysis The amount of data that can be encoded depends on three factors, (i) the spendable budget \( n \), (ii) the number of transaction outputs \( k \) and (iii) their corresponding lower bounds \((l_1, ..., l_k)\).

We can determine the degree of variability by evaluating

\[
\log_2 \binom{n' + k - 1}{k - 1}
\]

In the next step we will estimate the degree of variability by applying Stirling’s formula [16, p.102]

\[
n! \approx \left( \frac{n}{e} \right)^n \cdot \sqrt{2\pi n}
\]
to approximate the binomial coefficient

\[
\binom{n}{k} \approx n \cdot (n - 1) \cdots (n - k + 1) \approx \left(\frac{n - k}{2}\right)^k \cdot e^k \cdot \sqrt{2\pi k} \approx n \cdot \left(\frac{n - k}{2}\right)^k \cdot e^k \cdot \sqrt{2\pi k}
\]

(19)

and thereby obtain its natural logarithm

\[
\ln\left(\binom{n}{k}\right) \approx k \cdot \ln\left(\frac{n}{k} - 0.5\right) + k - 0.5 \cdot \ln(2\pi k)
\]

(20)

Finally after substituting the values of \(n\) and \(k\) as well as an appropriate base conversion, we get the formula for the degree of variability

\[
\log_2\left(\binom{n'}{k} + 1 \right) \approx \frac{(k - 1) \cdot \ln\left(\frac{n' + k - 1}{k - 1} - 0.5\right) + (k - 1) - 0.5 \cdot \ln(2\pi (k - 1))}{\ln(2)}
\]

(21)

As illustrated in Fig. 4.4, both the spendable budget \(n\) and the number of transaction outputs \(k\) play a significant role. The spendable budget is strongly bounded by the financial means of the user and therefore cannot be chosen arbitrarily. At the same time the costs of embedding data do not increase with the budget and thus it should be maximized to achieve the best degree of variability.

![Figure 4.4: Degree of Variability in Transaction Budget Splitting](image)

The number of transaction outputs is, compared to the spendable budget, much easier to maximize, since it can be chosen nearly at will. A major drawback is that each additional transaction output takes up additional space for which new costs arise and must therefore be chosen carefully.
Analysis

Budget Claiming
Aside from the possibility of embedding data in a combinatorial composition of the spendable budget, there is another way to improve the scheme. The transaction outputs, on the one hand, must be ordered since the composition is sensitive to permutations. The transaction inputs claiming them, on the other, can be permuted arbitrarily. We will use this degree of variability to embed information by using the method described in the following.

Method A vector of transaction outputs \((y_1, ..., y_k)\) can be claimed in \(k!\) different ways. Thereby we can encode an integer \(d\) with \(0 \leq d < k!\) as a permutation

\[
y_{\pi_d(1)}, \ldots, y_{\pi_d(k)}
\]

of the output indices, where the function \(\pi_d\), described in Sect. 4.3.3, computes the \(d\)-th permutation of the ordered vector \((1, ..., k)\). Note that a permutation \(\pi_d = (\pi_d(1), ..., \pi_d(k))\) maps the first element 1 to some other value \(\pi_d(1)\), the second element 2 to yet another value \(\pi_d(2)\) and so forth.

The starting point for the recovery of the permutation index \(d\) is a vector of transaction inputs from which we can directly extract the permuted index vector \((\pi_d(1), ..., \pi_d(k))\).

Once we have obtained the permutation \(\pi_d\), we retrieve the integer \(d\) by the method described in [17, p.19]:

\[
d = \sum_{i=1}^{k-1} a_i \cdot (k - i)!
\]

where \(a_i = \#\{\pi_d(i) > \pi_d(j) : k \geq j > i\}\) is the number of elements \(\pi_d(j)\) to the right of \(\pi_d(i)\) that are smaller than \(\pi_d(i)\). For more background information see the description provided in Sect. 4.3.3.

Analysis The degree of variability can be approximated by applying Stirling’s formula [16, p.102]

\[
k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}
\]

as follows

\[
\log_2 k! \approx k \cdot (\ln(k) - 1) + 0.5 \cdot \ln(2\pi k) - \ln(2)
\]

As presented in Fig. 4.5, thanks to this method, we are able to embed a significant amount of information at no additional cost.

4.2.4 Lock Time and Sequence Numbers

The transaction lock time and the transaction input sequence numbers are jointly constrained by the final transaction rule described in Sect. 3.2.3. The rule states that at least (i) the transaction lock time is set to locked or (ii) the transaction input sequence numbers are final. Since both fields are of the same size and there are always at least as many sequence number fields as there are lock time fields, it is preferrable to utilize
sequence number fields for data inclusion.

The value of sequence number fields can be chosen arbitrarily when the transaction lock time is set to locked and we thereby obtain a degree of variability of 32 bits per transaction input.
4.3 Mapping Functions

In this section three mapping functions will be introduced and discussed. These functions will be used to embed data into and to extract data from a pre-defined data structure.

4.3.1 Mapping of Strings to EC Points

In order to use ECC all parties within the system must agree on a specific elliptic curve and a set of associated domain parameters. The parameters associated with the Koblitz curve secp256k1, which is utilized by Bitcoin, are specified by the sextuple \((p, a, b, G, n, h)\) over the prime finite field \(\mathbb{F}_p\). The finite field \(\mathbb{F}_p\) is defined by

\[
p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
\]

whereas the elliptic curve \(E: y^2 \equiv x^3 + ax + b\) over \(\mathbb{F}_p\) is defined by

\[
a = 0 \\
b = 7
\]

The remaining parameters are the generator \(G\), the order \(n\) of \(G\) and its cofactor \(h\). The knowledge of their exact values is not necessary for the following analysis and will be skipped.

**EC Points**

A point \((x, y)\) on the elliptic curve is defined by its \(x\) and \(y\) coordinates in \(\mathbb{F}_p\). Furthermore, a point is said to be valid if it satisfies the curve equation \(E: y^2 \equiv x^3 + ax + b\) over \(\mathbb{F}_p\). For a given \(x\)-coordinate there are precisely 0, 1 or 2 corresponding \(y\)-coordinates that satisfy the curve equation.

Elliptic curve points are typically encoded using the ANSI X9.62 encoding scheme that has been shortly introduced in Sect. 4.2.1. In this scheme points are encoded either in a prefixed compressed form, where only the \(x\)-coordinate is stored, or in a prefixed uncompressed form, where both the \(x\) and \(y\)-coordinate are stored. For any compressed point the \(y\)-coordinate can be uniquely determined by \(\pm \sqrt{x^3 + ax + b}\), where the prefix indicates the sign. For this reason a mapping function to compressed points will be described.

The conditions stated above yield two essential requirements that must be fulfilled by our mapping function, (i) the \(x\)-coordinate value must be in \(\mathbb{F}_p\) and (ii) there is a \(y \in \mathbb{F}_p\) such that the pair \((x, y)\) satisfies the curve equation. Note, however, that not every \(x\) in \(\mathbb{F}_p\) will allow for a valid curve point. In the following we shall discuss this problem in more detail.

**Experiment**

The core of the problem lies in the unpredictable distribution of points on the curve. A feasible solution to this problem is to determine the maximal distance between two consecutive valid EC points and based on the results reserve an adequate amount of space within which values will be enumerated until a valid point is found. We shall formulate the experiment objective as follows:
Let the distance between two neighbouring points \( P_1 = (x_1, y_1) \) and \( P_2 = (x_2, y_2) \) be defined as

\[ |x_2 - x_1| \] (28)

where two points are defined as neighbours if there is no \( x_i \) with \( x_1 < x_i < x_2 \) s.t. \( P_i = (x_i, y_i) \) is a valid point on the curve. Note that \( y_i \) can be obtained from \( \pm \sqrt{x^3 + ax + b} \).

What is the maximum distance between two neighbouring points \( P_1 = (x_1, y_1) \) and \( P_2 = (x_2, y_2) \) on a given elliptic curve \( E \)?

This experiment has been conducted for a sample size of 100,000 randomly chosen points on the Koblitz curve secp256k1 and the results indicate that the maximum distance is 20.

**Mapping Function**

Define a mapping function \( \phi : \{0, 1\}^n \times \{0, 1\}^k \rightarrow E(\mathbb{F}_p) \) from a pair of \( n \leq 250 \) bit and \( k = 255 - n \) long strings to elliptic curve points by

\[ \phi(b_1...b_n, w_1...w_k) = (02_{16}||02||b_1...b_n||w_1...w_k) \] (29)

where \( b_1...b_n \) is an \( n \)-bit long string and \( w_1...w_k \) is a \( k \)-bit string, which is randomly iterated until \( (02_{16}||02||b_1...b_n||w_1...w_k) \) is a valid curve point.

We know that the \( x \)-coordinate \( (02||b_1...b_n||w_1...w_k) \leq 2^{255} < p \) due to the leading 0-bit and thus the first requirement is satisfied. Furthermore, the previously discussed experimental results indicate that for practical purposes we may safely assume that the trailing \( k \geq 5 \) (\( \lceil \log_2 20 \rceil \)) bits \( w_1...w_k \) provide sufficient variance to always find a valid point and thus the second requirement is satisfied as well.

The mapping function \( \varphi : E(\mathbb{F}_p) \rightarrow \{0, 1\}^n \) is trivially defined by

\[ \varphi(02_{16}||02||b_1...b_n||w_1...w_k) = b_1...b_n \] (30)

It is easy to see that for every EC point of the format \( (02_{16}||02||b_1...b_n||w_1...w_k) \) the string \( b_1...b_n \) can be retrieved.

**4.3.2 Mapping of Integers to k-Compositions**

A combinatorial composition is defined in [18, pp.60-62] as an ordered arrangement of \( k \) non-negative integers which sum up to \( n \). Put into mathematical terms, a \( k \)-composition of an integer \( n \) is defined by the equation

\[ x_1 + \cdots + x_k = n, \ x_i \in \mathbb{N} \text{ and } 1 \leq i \leq k \] (31)

The number of compositions \( f(n, k) \) for a \( k \)-composition of an integer \( n \) is given by

\[ f(n, k) = \binom{n + k - 1}{k - 1} = \binom{n + k - 1}{n} \] (32)
Furthermore, let \( S(n, k) \) denote the set of \( k \)-compositions of an integer \( n \) in lexicographic order. For example, \( S(3, 3) \) is given by
\[
S(3, 3) = \{003, 012, 021, 030, 102, 111, 120, 201, 210, 300\}
\] (33)
The problem we are trying to solve is, given an integer \( n \), a number of parts \( k \) and an index \( d \) with \( 0 \leq d < f(n, k) - 1 \), to determine \( S(n, k)[d] \), the \( d \)-th element of the list \( S(n, k) \).

**Analysis**

We construct the \( d \)-th element of \( S(n, k) \) iteratively, by first finding \( x_1 \), then \( x_2 \), and so forth. Once \( x_1 \) to \( x_i \) with \( 1 \leq i \leq k \) are known, the remaining components satisfy
\[
x_{i+1} + \cdots + x_k = n - \sum_{j=1}^{i} x_j
\] (34)
i.e. the same problem with reduced parameters. In the following we will present an efficient method, described in [19], to iteratively find each component \( x_i \) with \( 1 \leq i \leq k \) of \( S(n, k) \).

Let us first begin by observing that for an integer \( \mu \) with \( 0 \leq \mu \leq n \) the formula
\[
f(n, k) - f(n - \mu - 1, k)
\] (35)
yields the number of compositions where \( x_1 \in \{0, \ldots, \mu\} \). We shall use this formula to count the number of lexicographically ordered solutions where the first component begins with \( \{0, \ldots, \mu\} \).

If we consider the \( d \)-th lexicographic \( k \)-composition of \( n \), then we know that
\[
f(n, k) - f(n - x_1 - 1, k) > d
\] (36)
holds, since \( f(n, k) - f(n - x_1 - 1, k) \) counts the number of lexicographically ordered compositions including the \( d \)-th composition itself. Since our labelling of compositions begins at 0, the expression \( f(n, k) - f(n - x_1 - 1, k) \) is indeed greater than \( d \). Furthermore we know that
\[
f(n, k) - f(n - x_1, k) \leq d
\] (37)
holds, since there are at most \( f(n, k) - f(n - x_1, k) \) lexicographically ordered compositions before the \( d \)-th composition itself. Thus we obtain the relation
\[
f(n, k) - f(n - x_1, k) \leq d < f(n, k) - f(n - x_1 - 1, k)
\] (38)
or written more practically
\[
f(n - x_1 - 1, k) < f(n, k) - d \leq f(n - x_1, k)
\] (39)
Now, we essentially want to solve
\[
f(n, k) - d \approx f(n - x_1, k)
\] (40)
or equivalently put
\[
(f(n, k) - d) \cdot (k - 1)! \approx f(n - x_1, k) \cdot (k - 1)!
\] (41)

For simplicity, we define
\[
A \overset{\text{def}}{=} (f(n, k) - d) \cdot (k - 1)!
\] (42)

and use it to obtain
\[
A \approx f(n - x_1, k) \cdot (k - 1)!
\] (43)

which is equivalent to
\[
A \approx (n - x_1 + 1) \cdots (n - x_1 + k - 1)
\] (44)

By approximation we get
\[
A \approx (n - x_1 + k/2)^{k-1}
\] (45)

which yields the solution
\[
x_1 \approx n - (A^{1/(k-1)} - k/2)
\] (46)

Note that the approximation is good if \( A \) is large enough with respect to \( k \). Thus, we have obtained the solution for \( x_1 \) and can solve the reduced problem iteratively in an identical fashion.

**Mapping Function**

Define a mapping function \( \psi : \mathbb{N}^3 \to \mathbb{N}^k \) from a triple of integers to a \( k \)-vector of integers by
\[
\psi(n, k, d) = S(n, k)[d]
\] (47)

where \( S(n, k)[d] \) is the \( d \)-th lexicographically ordered \( k \)-composition of the integer \( n \). The composition is computed as follows:

**Algorithm 1:** Composition Computation

- **Input:** An integer \( n \), a number of parts \( k \) and a composition index \( d \)
- **Output:** Composition \((x_1, \ldots, x_k)\)

1. For \( i := 1\ldots k - 1 \) Do
   1.1 \( A \leftarrow (f(n, k) - d) \cdot (k - 1)! \)
   1.2 \( \mu \leftarrow n - \lfloor A^{1/(k-1)} - k/2 \rfloor \)
   1.3 While \( f(n - \mu - 1, k) \geq f(n, k) - d \) Do \( \mu \leftarrow \mu + 1 \)
   1.4 While \( f(n, k) - d > f(n - \mu, k) \) Do \( \mu \leftarrow \mu - 1 \)
   1.5 \( n \leftarrow n - \mu \)
   1.6 \( k \leftarrow k - 1 \)
   1.7 \( d \leftarrow d - (f(n, k) - f(n - \mu, k)) \)
   1.8 \( x_i \leftarrow \mu \)

2. \( x_k = n \)
3. Return \((x_1, \ldots, x_k)\)
Reversing the Mapping

We can reconstruct the index \( d \) from a composition \((x_1, \ldots, x_k)\) by simply evaluating

\[
d = \binom{n + k - 1}{k - 1} - 1 - \sum_{i=1}^{k-1} \left( \sum_{j=k-i+1}^{k} x_j + i - 1 \right)
\]  

(48)

Note that the derivation of this formula is described in more detail in Appendix C. Interestingly, according to [20], this computation is connected with the unique integer representation theorem mentioned in [17, p.8], since

\[
\sum_{i=1}^{k-1} \left( \sum_{j=k-i+1}^{k} x_j + i - 1 \right) = \sum_{i=1}^{k-1} \left( \sum_{j=1}^{k} y_j + i - 1 \right)
\]

(49)

where \( y_j = x_{k-j+1} \), is equivalent to the unique integer representation theorem

\[
L = \sum_{i=1}^{k-1} \binom{q_i}{i} \quad \text{with } q_{i+1} > q_i \geq 0
\]

(50)

for the special choice of

\[
q_i = \sum_{j=1}^{i} y_j + i - 1
\]

(51)

4.3.3 Mapping of Integers to Permutations

We begin by stating that, according to [17, p.7], at given \( k \) each integer \( d \) with \( 0 \leq d < k! \) has a unique factorial representation

\[
d = \sum_{i=1}^{k-1} a_i \cdot (k - i)! \quad 0 \leq a_i \leq k - i
\]

(52)

where the \( a_i \)'s are referred to as factorial digits of \( d \) and \((a_1, \ldots, a_{k-1})\) is its factorial representation with a falling factorial base. On the basis of this statement we shall establish in the following a one-to-one correspondence between the factorial representation of integers and permutations. Once that correspondence is found, we will be capable of lexicographically labelling permutations.

Computing the Factorial Representation

Let \( d \in \{0, \ldots, k! - 1\} \) be an integer, then its corresponding factorial representation is given by the recursive relation

\[
a_i = \left\lfloor \frac{d - \sum_{j=1}^{i-1} a_j \cdot (k - j)!}{(k - i)!} \right\rfloor \quad \text{for } i = 1 \ldots k - 1
\]

(53)

For example, the integer \( d = 15 \) with \( k = 4 \) yields the factorial representation

\((2, 1, 1)\)

(54)
Computing the Permutation
From the factorial representation \((a_1, ..., a_{k-1})\) we can then obtain the permutation \((\pi_d(1), ..., \pi_d(k))\) as
\[
\pi_d(i) = y_i[a_i + 1] \text{ for } i = 1 \ldots k - 1
\]
and \(\pi_d(k) = y_k[1]\), where \(y_i = y_{i-1} \setminus y_{i-1}[a_{i-1} + 1]\) and \(y_1\) is a vector with \(k\) elements in ascending order of which the permutation is required.

For the previous example with the factorial representation \((2, 1, 1)\) and the ordered vector \((1, 2, 3, 4)\) this would yield the permutation
\[
(3, 2, 4, 1)
\]

Reversing the Process
Finally, the integer \(d\) can be retrieved for a given permutation \((\pi_d(1), ..., \pi_d(k))\) by
\[
d = \sum_{i=1}^{k-1} a_i \cdot (k - i)! , \ 0 \leq a_i \leq k - i
\]
where \(a_i = \# \{ \pi_d(i) > \pi_d(j) : k \geq j > i \}\) is the number of elements \(\pi_d(j)\) to the right of \(\pi_d(i)\) that are smaller than \(\pi_d(i)\). For example, for the permutation \((3, 2, 4, 1)\) we obtain the factorial representation
\[
(2, 1, 1)
\]
which is then used to get the label
\[
d = 2 \cdot 3! + 1 \cdot 2! + 1 \cdot 1! = 15
\]
This gives us a one-to-one correspondence between integers and permutations which can now be utilized to encode integers as permutations.
4.4 Experiments
In the following a set of complementary experiments will be presented that will supplement the previous analysis with additional information and confirm its correctness. In fact, it is necessary that they are conducted. There are significant differences in node behaviour that stem from the fact that each individual node possesses (i) a specific protocol support and (ii) a specific node policy. It is therefore necessary to determine whether any unanticipated restrictions exist by means of trial-and-error.

4.4.1 Inclusion of Invalid Public Keys
The purpose of this experiment is to verify whether the use of invalid public keys is permitted. We define an invalid public key as a 33 bytes or 65 bytes long arbitrary string.

Motivation
Invalid public keys yield a higher data density compared to valid public keys. Furthermore they simplify the data inclusion process since they do not require a dedicated function for mapping of strings to valid elliptic curve points.

Results
We conducted the experiment for two standard transaction types, P2PK and P2SH P2PK, to ensure that no validity checks are performed on public keys in both signature and public key scripts.

The first part of the experiment was performed for transactions of P2PK transaction type and its results can be found in the transaction with the transaction ID 52e307670ad54db6490f8ac11fe456a71dd0ef89fbd152b1248f25eb69bb504 which embeds the invalid public key

2b 110000072f52321f7573722f62696e2f707974686f6e0a0a46696e6973656e74696f6e20746f6f6c206f7220426974636f696e0a2320

The second part of the experiment was performed for transactions of P2SH P2PK transaction type and its results can be found in the transaction with the transaction ID fc337be8adac9dc08e89d58a8474e2022d2e80f8dfe1b186f204e626e2b106df which embeds the invalid public key

04 11000083a072f52321f7573722f62696e2f707974686f6e0a0a46696e6973656e74696f6e20746f6f6c206f7220426974636f696e0a2320

In both cases we were able to include invalid public keys and we therefore conclude that public keys do not undergo any validity checks.

4.4.2 P2SH Multisig with 1-of-12 Public Keys
The goal of this experiment is to confirm that it is indeed possible to create P2SH Multisig transactions with one signature and twelve public keys.
Motivation
The analysis of degrees of variability in various standard transaction types has shown that P2SH Multisig achieves the best results. It is therefore important to ensure that there are no unpredicted restrictions before an embedding scheme is developed.

Results
The results show that it is possible to create P2SH Multisig transactions with one signature and twelve compressed public keys. It can be found in the transaction with the transaction ID

\text{d5e02512d5bb3c44267030fc556695ef4802ab9fa825d6a50be6d8b2083a64f9}

4.4.3 Nulldata Limitations
The objective of this experiment is to determine the support for Nulldata transactions by miners.

Motivation
The Nulldata transaction type has been recently introduced in the reference implementation [21] and we thus anticipate that it will only be accepted by a fraction of miners.

Results
The results indicate that only a fraction of miners supports Nulldata transactions. The transaction with the transaction ID

\text{14a746118b0079407dd0e42d6ae1938f47cc51ee4eb430dd52e3b25fad32f665}
was submitted on the 16.6.2014 at 14:01 and included in a block on the 16.6.2014 at 15:42. At the time a standard transaction with 0.0001 BTC was typically included within 10 minutes. Hence, due to the low support there is a significant delay that should be taken into account.

4.4.4 Chained Transaction Delays
The aim of this experiment is to ascertain how quickly transaction outputs can be spent.

Motivation
The rate at which transaction outputs can be spent has an impact on how quickly chained transactions and therefore the included data can be published.

Results
The results show that it is possible to spend transaction outputs immediately within the same block. This can be observed in the two transactions with the transaction IDs

\text{686b8ae76ddcd2cc039600c384a8dc398945cd8ce89b3f776e702cf06f4c1ad0}
\text{0014fba8752fe496b39c2508b14c1d62ca5a26b4b08b45844eef7278a816901}
which have been included in the same block. A chain of ordered transactions can thereby be submitted in one go and at best be included in the next block.
4.5 Data Compression

In the following we will present a sample data compression technique that can be used to reduce the overall size and therefore the costs of the included data. We will briefly describe the compression scheme, evaluate it and discuss the results.

4.5.1 Huffman Coding

The Huffman code, named after its creator David Huffman, is a prefix code commonly utilized for lossless data compression. It is optimal among all symbol-by-symbol coding techniques with a known symbol probability distribution and achieves best results if these are negative powers of two.

The coding process can be divided into two steps, the calculation of a symbol probability distribution for a given text and the computation of the corresponding codewords.

4.5.2 Evaluation Framework

Before we shall evaluate the Huffman coding technique, we improve the performance of the scheme by restricting the alphabet to the following set of characters:

1) Lowercase Latin Alphabet
   a b c d e f g h i j k l m n o p q r s t u v w x y z
2) Uppercase Latin Alphabet
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
3) Digits
   0 1 2 3 4 5 6 7 8 9
4) Punctuation Marks
   ? ! ’ ” - , : ; ( ) / 
5) General Typography
   * + @ < > = # % & $
6) Auxiliary Delimiters
   Space, Newline and End-of-File characters

We will derive the probability distribution for these symbols from a large set of structured texts, referred to in linguistics as a text corpus. For this purpose we have utilized the Open American National Corpus (OANC) [22], which is a large and publicly available collection of various texts in American English. Note that for these texts all characters that are not in the specified alphabet are excluded.

4.5.3 Evaluation Results

For the performance evaluation of the coding technique we have (i) picked a random subset of 8000 files from the OANC to generate the symbol probability distribution and the corresponding codewords and (ii) used the remaining 806 files to assess the achieved data compression ratio. The results yield an average data compression ratio of 1.79955 or, equivalently put, space savings of approximately 44.43%.
4.5.4 Discussion

There are several points regarding the choice of the compression technique and the evaluation results that we would like to stress. Firstly, the employed compression technique is only supposed to illustrate the possibility of reducing the data embedding costs by means of data compression. The Huffman coding method is, compared to other contemporary compression techniques, relatively primitive. As shown in [23], some of the best contemporary compression algorithms for texts utilize Context Mixing (CM), Prediction by Partial Match (PPM) and Burrows Wheeler Transform (BWT) as their underlying compression technique, with much better results.

Secondly, data compression algorithms perform differently for texts of (i) different lengths as well as (ii) different languages. The text length can have a significant impact since some algorithms only perform well when sufficient text is provided and until that threshold they might even produce negative compression results. Furthermore many algorithms utilize dictionaries for compact encoding of common sequences of symbols. These dictionaries are based, analogously to the probability distribution in the Huffman coding, on a specific text corpus. This text corpus, in turn, is written in a specific language which by implication determines the performance of the algorithm.

Finally we have subtly assumed that data compression will be performed on texts only. However, it is perfectly feasible to embed any type of data for which either no compression at all or a compression method adequate for the given data type must be chosen.
4.6 Data Embedding Model

In the previous sections we have inspected various parts of a regular transaction under the aspect of suitability for embedding data. We will now interpret these results to extract existing parameters and determine their optimal values for a data embedding model. Finally, we will discuss the results in more detail.

4.6.1 Preliminary Considerations

Before we can proceed with constructing the model, we must first consider the scope of our evaluation. As we know, a transaction possesses a chained structure where transaction outputs and inputs, together with their corresponding public key and signature scripts, exist in pairs. The previous analysis has shown conclusively that the these script pairs are well suited for data embedding purposes. Hence the embedded data spans several transactions instead of a single one and we must make certain assumptions in order to simplify the analysis.

\[ \text{Figure 4.6: Scope of Data Embedding Model} \]

For the analysis we will assume an infinitely long stream of data that is to be embedded into an infinitely long chain of transactions. The embedded data will, for any intermediary transaction in the chain, overlap between the current transaction and its successor. We have therefore decided to take into consideration, as depicted in Fig. 4.6, all fields of the current transaction, except for the transaction input counter and the transaction inputs, which are taken from the succeeding transaction. Thereby we can reduce any two consecutive transactions in the chain into one and perform the parameter optimization on it.

4.6.2 Evaluation Framework

The previous sections have shown which parts of regular transactions are fixed and which parts are variable. The variable parts are of particular interest, since these have
an impact on the embedded data size and the costs. We have determined the following set of parameters:

- $\alpha$: The number of P2SH Multisig transaction input-output pairs
- $\beta$: The number of Nulldata outputs
- $\gamma$: The available budget in Satoshi

In order to determine the optimal values for these parameters, we construct a cost function and an embedded data size function, and then optimize the rate between them as:

$$\min_{\alpha, \beta, \gamma} \frac{\text{Cost}(\alpha, \beta)}{\text{Data}(\alpha, \beta, \gamma)} \quad (60)$$

The rate between the two functions yields the cost rate per embedded byte which we attempt to minimize. Note that we constraint the values for the parameters by $1 \leq \alpha \leq 20$, $0 \leq \beta \leq 1$ and $0 \leq \gamma \leq 10^8$.

### 4.6.3 Results

The results of the parameter optimization have shown that in order to achieve the best cost rate the data embedding model should use (i) the complete budget of 1 Bitcoin, (ii) 14 times P2SH Multisig transaction input-output pairs and (iii) once a Nulldata transaction output. With these parameters we achieve a cost rate of approximately 16 Satoshis per embedded byte.

![Figure 4.7: Cost Rate Scaling](image-url)

There are three noteworthy points about these results. Firstly, the impact of the available budget on the cost rate of embedded data is negligible. This is illustrated above in Fig. 4.7, where even a hundredfold increase in the available budget does not yield
any noticeable improvements. In fact, the aforementioned increase improves the cost rate only by approximately 0.04 Satoshi per embedded byte and therefore leads to an overlap of the plots.

Secondly, an increase in the number of elements that embed data, such as the P2SH Multisig transaction input-output pairs, does not necessarily yield an improved cost rate. As shown in Fig. 4.7, the cost rate fluctuates significantly for small transaction sizes and slowly stabilizes for bigger transaction sizes. This behaviour can be attributed to two factors, the relative size of embedded data within a transaction and the transaction fee function.

![Figure 4.8: Transaction Fee Scaling](image)

As illustrated above in Fig. 4.8, since the transaction fees are calculated per 1000 bytes, they scale in a non-linear fashion. The best cost rate is therefore achieved when both the rate between the embedded data size and the total transaction size is highest, and when the total transaction size is just beneath a multiple of 1000 bytes.

Finally, the cost rate swiftly converges towards its optimum which lies, at the current transaction fee rate, only marginally beneath the achieved 16 Satoshi per embedded byte. Considering that the current price per Bitcoin lies around 300 Euro, we obtain an average cost rate of 4.8 Cent per 1000 embedded bytes.
5 Software Description

In this section we focus on the practical part of the thesis, the proof of concept implementation. We recapitulate the requirements and discuss its final design.

5.1 Requirements

In Sect. 1.2, we have discussed that the core functionality of the system should comprise:

i) Writing messages into the blockchain under a pseudonym

ii) Reading messages published in the blockchain under a pseudonym

Aside from these two functional requirements, there is another constraint, namely the system must be an extension of the existing Bitcoin software. This enforces a modular architecture of the software which allows any user to deploy and, if necessary, to remove the messaging system without any adverse effects on the pre-installed Bitcoin software.

5.2 Design

The design will be described from a static and a dynamic point of view. In the static view we will visualize the overall architecture of the system, its components and their interaction. In the dynamic view, on the other hand, we will visualize the workflow of the software. In particular, the steps in the process of embedding messages into as well as extracting messages from the blockchain will be described.

5.2.1 Architecture

The system architecture comprises, as depicted below in Fig. 5.1, three components - the Bitcoin Messaging System, the Bitcoin API wrapper and the Bitcoin Daemon. In the following we will discuss each of these components in more detail.

![Figure 5.1: System Architecture](image-url)
**Bitcoin Daemon**
The Bitcoin daemon is a piece of software that implements the Bitcoin protocol, a Bitcoin wallet and a JSON-RPC interface. The daemon is the core component of the Bitcoin software that can be downloaded from the official Bitcoin website [24].

Essential tasks of the Bitcoin protocol are, in this context, the maintenance of the block-chain and the support of auxiliary functions for interaction with the blockchain as well as the Bitcoin network.

The Bitcoin wallet is a separate piece of software that utilizes the Bitcoin protocol to implement, similarly to ordinary wallets, a system to manage existing funds. In practice several implementations of wallets exist and can be used interchangeably.

The JSON-RPC interface offers a comprehensive set of functions to manage the local wallet as well as to operate the Bitcoin protocol. It is used in the context of this system to establish a communication channel between the separate components and thereby achieve the desired architectural modularity.

Note that the Bitcoin daemon is assumed to be pre-installed and appropriately configured on the system where the messaging system is to be deployed. We require that it is configured such that the JSON-RPC interface is activated and credentials are configured as well as that the full transaction index is built, both of which can be done with little effort.

**API Wrapper**
This component is a C++ wrapper library for JSON-RPC communication with the Bitcoin daemon. It facilitates communication by converting messages into a JSON-RPC compatible format.

This library had to be developed from scratch for the purpose of this project and was made available to the public on GitHub [25]. We have found that although one alternative exists [26], the project was abandoned a long time ago and the code contains numerous bugs.

**Bitcoin Messaging System**
The last component is the messaging system itself. It implements the core functionality to create transaction chains that embed messages and, once they have been accepted into the blockchain, to extract them.

The process of embedding and the process of extracting data in transaction chains requires functionality provided by the Bitcoin daemon. In particular, the wallet is used to pay for transaction chains whereas the auxiliary functions in the Bitcoin protocol are used to send transaction chains into the Bitcoin network and, after miners have included them in the blockchain, to extract them.
5.2.2 Workflow

As mentioned before, there are two core functions that are implemented by the messaging system, the embedding of messages into the blockchain and their consecutive extraction from the blockchain. In the following we will describe both functions in more detail.

Embedding messages

The process of embedding a message begins with reading it. Next it is compressed and the embedding costs are estimated. The estimate is computed by building a transaction chain embedding the compressed data and calculating from it the transaction costs. After verification that the wallet holds sufficient funds, the process is continued.

In the next step the compressed data is embedded in a transaction chain. As the process is fairly complex, only a rough outline will be given. Firstly, two transactions are created and a pointer is set to the first transaction. Secondly, the optimal parameters are calculated and the compressed data is embedded into the transaction in the following order - P2SH Multisig transaction input-output pairs, the Nulldata transaction output and finally the budget split and budget claim. Note that the data will span over both transactions due to the inherent nature of scripts. Finally, if there is any remaining data, then another transaction is added, the pointer is incremented and the last step is repeated.

![Diagram of Transaction Structure with Embedded Data](image)

The resulting structure after embedding data resembles the transaction chain depicted in Fig. 5.2. The first transaction has one transaction input that supplies the chain with funds and marks the beginning of the chain. Then, the embedded data spans from the transaction outputs of the first transaction to the transaction inputs of the last transaction. Finally, the last transaction has one transaction output that bundles the remaining funds and marks the end of the chain.

During the embedding of messages into a transaction chain a trick is applied to ensure chaining between messages from the same sender. After the first message has been sent,
the last transaction output is memorized by the messaging system and used as the first transaction input in the next transaction chain. This way transaction chains and therefore messages are linked together and can, given a starting point, be read chronologically.

Once the transaction chain is constructed, the user is requested to enter the wallet password. If it is correct, then the transaction chain is signed and sent into the network. Finally, a message identifier is printed and can be used to retrieve a message once the transaction chain has been accepted into the blockchain. More information on message identifiers can be found in Sect. 5.2.3.
Figure 5.3: Message Publishing Process Flow
Extracting messages
The process of extracting messages from the blockchain begins with reading a message identifier. The message identifier is then used to find the transaction chain in which the message is located. How this is done depends on the operation mode and is explained in more detail in Sect. 5.2.3.

Once the transaction chain is retrieved, the remaining steps are straightforward. The compressed message is extracted from the transaction chain in the same order it was embedded and then decompressed. At this point the message is fully recovered and printed out.

Figure 5.4: Message Extraction Process Flow
5.2.3 Operation modes

There are currently two operation modes, a forward search and a backward search mode, in which the messaging system can be run. The chosen mode has an impact on how messages are identified and retrieved from the blockchain.

Before we dive into explaining the operation modes, however, it is important that we clarify the used terminology. Transaction chains are linked transactions that embed a single message. Message chains, on the other hand, are linked transaction chains and therefore embed several messages.

**Forward Search**

In the forward search mode the message identifier is the transaction ID of the beginning of a message chain. Upon supplying the identifier, all messages along the message chain are fetched and printed out.

The search begins by finding the block in which the supplied transaction ID resides. Since the Bitcoin daemon manages a transaction index that allows this, the task is trivial. The next transaction is found by inspecting whether the current block contains a transaction that refers to the previous transaction. If not, then the next block is inspected. The process is repeated until the end of the transaction chain is reached. Note that the end of a transaction chain is reached when it has precisely one output of a standard transaction type distinct from those utilized in intermediary transactions.

At this point the first transaction chain, and thereby the first message, has been successfully recovered. The recovery of consecutive messages, however, is more difficult. The next transaction chain, if it exists, may be located shortly after the last one or much farther in the blockchain. Therein lies the problem, due to Bitcoin’s architecture, it is technically impossible to quickly locate the next transaction in a chain. This would require to either compute a look-up table solely for this purpose, which due to the sheer amount of transactions is a hard problem, or search naively through the blockchain, as it was done above, which is a hard problem as well.

Despite these limitations, the forward search mode is particularly suitable for the purpose of running a newspaper service, where the identifier is used to identify and retrieve all messages ever published by the service. The only difficulty then lies in obtaining an authentic identifier.

**Backward Search**

In the backward search mode the message identifier is a pair of two transactions IDs that represent the beginning and the end of a message chain. Analogously to the previous mode, upon supplying the identifier, all messages along the message chain are fetched and printed out. The only difference lies in that the messages until the supplied end of the chain are fetched.
The search begins by finding the block including the transaction with the given transaction ID symbolizing the end of the message chain. Once again, since Bitcoin manages a transaction index that allows this, the task is trivial. Next, we look up the preceding transaction in the chain and fetch it. This process is repeated until the whole transaction chain is retrieved. In contrast to the forward search, looking up the preceding message is a trivial task, since looking up preceding transactions is done efficiently.

Generally speaking the forward search mode is preferable over the backward search mode. Considering the same scenario, a news service, in the forward search mode it was sufficient for the reader to obtain a single identifier and from then on passively wait for new messages whereas in the backward search mode the reader must actively receive a new identifier with the updated message chain end in order to read new messages. This can, to our knowledge, only be done efficiently with the help of online services, such as [27], which store extensive databases through which the required information can be extracted.

Note that the backward search mode is currently activated in the messaging system by default due to stability issues with the Bitcoin daemon when using the forward search mode.
6 Conclusions

The central focus of the thesis was to build an anti-censorship tool based on Bitcoin. The work consisted of two parts, a theoretical part where a Bitcoin protocol specification had to be devised and analysed as well as a practical part where, based on the analysis in the theoretical part, a proof of concept of an anti-censorship tool had to be implemented. Here we conclude our work by first giving a brief summary, then describing encountered issues and our contributions, and lastly discussing future work.

6.1 Summary

The thesis began with a literature study to identify existing work on embedding data in the blockchain. We briefly evaluated various approaches and concluded that these contain considerable shortcomings. Not only were these cost-inefficient but also functionally inadequate for an anti-censorship tool.

We began by introducing a formal description of the Bitcoin protocol in order to create a solid theoretical framework for the design of our anti-censorship system. We described the fundamental components of Bitcoin’s architecture, namely blocks and transactions as well as its peculiar constructs, such as the various transaction types, signatures and addresses. In particular, we focused on the structure of blocks and transactions, and the dependencies that exist within them.

Subsequently we used the protocol description to determine which parts of transactions are suitable for data embedding. For this purpose we first classified which fields can be sufficiently controlled by the user and then ascertained how much information can be embedded in them. For fields where data could not be embedded directly, special mapping functions were constructed. Additionally, we illustrated the possibility of using data compression to reduce the overall data size and thereby the costs. The results show that even basic compression techniques, such as Huffman coding, can effectively reduce the data size by up to 45%.

Furthermore we described a sample model for data embedding and optimized its parameters to reduce the costs even further. The optimization has shown that the optimal cost rate lies, at the current transaction fee rate, at around 16 Satoshi per embedded byte. Considering the current price of 300 Euro per Bitcoin, this is equivalent to 4.8 Cents per 1000 bytes. A cost rate marginally below the optimum can be achieved by utilizing a regular transaction with 14 P2SH Multisig transaction input-output pairs and 1 Nulldata output. The budget had no significant impact on the results.

Finally, we implemented a fully functional proof of concept and described its underlying design. The design description explained the system’s architecture, its workflow and its limitations.

6.2 Project Progress

The project progress has varied greatly throughout its duration due to a wide variety of reasons. In the following we will document the issues encountered in the different stages...
The first stage of the project comprised preliminary research and the development of a Bitcoin protocol specification. Against our expectations, it has emerged to be a truly time-consuming task. Despite the exceptional popularity of Bitcoin, and the wide and frequent press coverage it has received, it came to us as a big surprise that there was no adequately detailed protocol description. It seemed increasingly irrational that so many have trusted this new currency and invested considerable amounts of money in it whilst, except for some committed developers, few knew how it actually works.

The circumstances explained before have forced us to start from scratch. The protocol specification was therefore developed by reverse-engineering the existing code base. The original code has changed significantly over the years to accommodate new requirements and include countless security improvements. For this reason the codebase version 0.9.0 was used, where the relevant parts comprise approximately 40,000 lines of code. Over a time period of approximately 5 months the specification was reconstructed until a satisfactory result was reached. The result was scrutinized by the general public as well as, by personal request, several Bitcoin core developers and it was approved as an official reference on Bitcoin.org [28].

The next stage of the project involved the analysis of the aforementioned specification to identify transaction elements in which arbitrary data can be embedded. During the analysis we encountered a difficulty, namely that we could not find algorithms to solve certain combinatorial problems. In particular, we required two algorithms, one to compute a chosen lexicographically-ordered combinatorial composition, and one to compute a chosen lexicographically-ordered permutation. To the best of our knowledge, no algorithms exist for either computation. For this reason we had to construct these on our own.

During the last stage of the project, the implementation of the anti-censorship tool, we ran into two problems. The first one was the lack of a C++ library that would allow us to establish a communication channel between arbitrary software and the Bitcoin API. Therefore, we have developed a library of our own that implements this functionality. Furthermore, in order to prevent others from doing the same work all over again, we made it available on GitHub [25]. The second problem was the lack of arbitrary-precision integer support in basic combinatorial functions implemented in the popular Boost libraries. We resolved this problem by simply re-implementing these functions.

6.3 Contributions

The project has produced three main contributions, (i) a Bitcoin protocol specification, (ii) a complete analysis of the protocol under the aspect of data embedding and (iii) a functional implementation of an anti-censorship tool.

The Bitcoin protocol specification is, in our opinion, one of the most important contributions of our work. In the beginning of the project, we were unaware that there
was no existing formal documentation of the Bitcoin protocol. The design documents of Bitcoin, if any ever existed, are unknown and any descriptions we found in online literature were vague, unstructured or incomplete. To our knowledge, this document provides the most complete and detailed treatment of the topic.

The analysis of the Bitcoin protocol under the aspect of data embedding is the first one of its kind. It provides a complete list of elements that may carry arbitrary data as well as definitions of auxiliary mapping functions. As such, it has not only formed the basis for our own anti-censorship tool, but it can also be used to construct future anti-censorship tools based on Bitcoin.

We believe that our proof of concept is the first fully functional and practical implementation of an anti-censorship tool based on Bitcoin. Security, simplicity and cost efficiency played an important role in the design of the tool. All aforementioned aspects facilitate the use of our anti-censorship tool in practice. It allows a user to establish a permanent and secure one-way communication channel, which can be used for example to run a reliable news service. Additionally, the communication channel can be set up and sustained against the lowest possible cost for this system.

6.4 Future Work
The results of this work have demonstrated that it is feasible to build an anti-censorship tool based on Bitcoin. However, the idea can be elaborated further in a number of ways.

Improved Implementation
In the scope of this project it was sufficient to construct a proof of concept of an anti-censorship tool. However, before it can be deployed in practice, it must be changed appropriately.

Firstly, the issues concerning the forward search mode must be resolved. In the current state the user must bear long waiting times when reading messages or switch to the less practical backward search mode.

Secondly, the implementation can be extended with a graphical user interface in order to increase usability. At the moment only a command line interface is provided and its use is to many user not particularly appealing.

Finally, the source code can be compiler and packaged for different platforms to simplify the installation and removal process. Currently in order to use the software, the user must manually compile it from the source code, a process that most will not be familiar with.

Security Evaluation
The most important aspect that should be elaborated further in the future is the system’s security. In order for the system to be used in practice, it is important to know
which guarantees it can provide, and, perhaps even more important, which ones it cannot. A few examples of security aspects that need to be researched are the extent to which the system’s functionality can be subverted and under which circumstances it is safe to use. Note that these are only suggestions and many more aspects need to be covered.
Acknowledgments

I would like to express my sincere gratitude to the people that have supported me throughout this graduation project. Firstly, I would like to thank my supervisor Dr. Boris Škorić, for his commitment, his incessant willingness to help and the countless excellent discussions. Secondly, I would like to thank Dr. Aart Blokhuis for his help on the subject of combinatorial compositions. Lastly, my honest thanks to the members of the Bitcoin community who have helped to shape the technical documentation of Bitcoin.
Appendix A  Data types

General data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>Signed integer in little-endian.</td>
</tr>
<tr>
<td>uint</td>
<td>4</td>
<td>Unsigned integer in little-endian.</td>
</tr>
<tr>
<td>uint8_t</td>
<td>1</td>
<td>Unsigned integer.</td>
</tr>
<tr>
<td>uint16_t</td>
<td>2</td>
<td>Unsigned integer in little-endian.</td>
</tr>
<tr>
<td>uint32_t</td>
<td>4</td>
<td>Unsigned integer in little-endian.</td>
</tr>
<tr>
<td>uint64_t</td>
<td>8</td>
<td>Unsigned integer in little-endian.</td>
</tr>
<tr>
<td>uint160</td>
<td>20</td>
<td>Unsigned integer array uint32_t[] of size 5. Used for storing RIPEMD160 hashes as a byte array.</td>
</tr>
<tr>
<td>uint256</td>
<td>32</td>
<td>Unsigned integer array uint32_t[] of size 8. Used for storing SHA256 hashes as a byte array.</td>
</tr>
</tbody>
</table>

Variable length integers (VarInt)

Integers in Bitcoin can be encoded depending on the value in order to save space. Variable length integers always precede vectors of a type of data that may vary in length. An overview of the different variable length integers is depicted below.

<table>
<thead>
<tr>
<th>Value interval</th>
<th>Size (Bytes)</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0, 2^8 - 3])</td>
<td>1</td>
<td>uint8_t</td>
</tr>
<tr>
<td>([2^8 - 3, 2^{16})]</td>
<td>3</td>
<td>0xFD followed by the value as uint16_t</td>
</tr>
<tr>
<td>([2^{16}, 2^{32})]</td>
<td>5</td>
<td>0xFE followed by the value as uint32_t</td>
</tr>
<tr>
<td>([2^{32}, 2^{64})]</td>
<td>9</td>
<td>0xFF followed by the value as uint64_t</td>
</tr>
</tbody>
</table>
Appendix B  Formulas

Transaction Fees
The transaction fees $TxFee$ in Satoshi are calculated from the transaction size $TxSize$ in bytes and the transaction fee rate $TxFeeRate$ in Satoshi per kB as follows:

$$TxFee = TxFeeRate \cdot \left\lceil \frac{TxSize}{1000} \right\rceil$$ (61)

The transaction fee rate currently lies at 10000 Satoshi per kB and can, if desired, be changed by the user. Note, however, that if it lies below the minimum transaction fee rate of 1000 Satoshi per kB, then it will neither be relayed nor mined.

Dust Transactions
A transaction is defined as “dust”, if any of the transaction outputs spends more than 1/3rd of its value in transaction fees. More precisely, a transaction is considered “dust” if any of its transaction outputs satisfies the inequality

$$\frac{MinTxFeeRate}{1000} \cdot (TxOutSize + 148) > \frac{1}{3}$$ (62)

where $nValue$ is the transaction output value, $TxOutSize$ is the transaction output size in bytes and $MinTxFeeRate$ is the minimum transaction fee rate in Satoshi per kB.

Minimum Spending Amount
It follows from the dust transaction rule that every transaction output has a minimum spending amount defined by

$$MinValue = 3 \cdot \frac{MinTxFeeRate}{1000} \cdot (TxOutSize + 148)$$ (63)

Given that the default minimum transaction fee rate currently lies at 1000 Satoshi per kB and that the size of a typical transaction output is 34 bytes, the resulting minimum spending amount is 546 Satoshi.
Appendix C  Derivations

Lexicographic index of k-compositions
In the following we explain how the index $d$ for a given $k$-composition of an integer $n$ can be derived. First, we define

$$d_{\text{max}} - d$$

as the number of compositions after the $d$-th composition. We know that this is computed by

$$f(n - (x_1 + 1), k) + f(n - x_1 - (x_2 + 1), k - 1) + \ldots$$

where the first expression counts the number of compositions with a larger $x_1$, the second expression counts the number of compositions with a larger $x_2$ and so forth. Note that it is sufficient to use $k - 1$ components to uniquely identify a composition.

We transform the previous equation into a more practical representation

$$\sum_{a=1}^{k-1} \left( n - \sum_{j=1}^{a} x_j + k - a - 1 \right)$$

(65)

Now if we substitute for $a = k - i$ we obtain

$$\sum_{i=1}^{k-1} \left( n - \sum_{j=1}^{k-i} x_j + i - 1 \right)$$

(66)

which is due to $\sum_{i=1}^{k} x_i = n$ equivalent to

$$\sum_{i=1}^{k-1} \left( \sum_{j=1}^{k-i+1} x_j + i - 1 \right)$$

(67)
Appendix D  ECDSA Signature Scheme

The ECDSA signature scheme can be seen as an elliptic curve variant of the widely known DSA signature scheme. It has two variable parameters [13], an elliptic curve with specific domain settings and an auxiliary hash function. In the context of Bitcoin, the Koblitz Curve secp256k1 with parameters defined in the Standards for Efficient Cryptography (SEC) [14] and double-SHA256 are used respectively. In the following a detailed description of the scheme, as given in [29, pp.282–285], will be presented.

Key Generation
Prior to the key generation process itself the system must be initialized with a set of parameters. An elliptic curve $E$ has to be chosen with a modulus $p$, coefficients $a$ and $b$ as well as a generator $G$ that generates a cyclic group of order $n$. The keys are then generated as follows:

1) Choose random integer $d$ with $0 < d < n$
2) Compute $A = dG$

The resulting public key $K_{pub}$ and private key $K_{pr}$ are

$$K_{pub} = (p, a, b, q, G, A)$$
$$K_{pr} = (d)$$

Signature Generation
Analogously to the DSA signature scheme, the signature consists of a tuple of integers $(R, S)$, each of the same bit length as $n$. The signature for a message $m$ is computed as follows:

1) Choose an integer as a random ephemeral key $K_E$ with $0 < K_E < n$
2) Compute $r ≡ K_E \cdot G$ (mod $n$)
3) Let $R = r_x$ be the $x$-component of $r$
4) Compute $S ≡ (h(m) + d \cdot R) \cdot K_E^{-1}$ (mod $n$)

Signature Verification
A signature $(R, S)$ for a given message $m$ is then verified as follows:

1) Compute auxiliary value $w ≡ S^{-1}$ (mod $n$)
2) Compute auxiliary value $u_1 ≡ w \cdot h(m)$ (mod $n$)
3) Compute auxiliary value $u_2 ≡ w \cdot R$ (mod $n$)
4) Compute $P ≡ u_1G + u_2A$ (mod $n$)
5) The verification $ver(m, (R, S))$ follows from:

$$x_P \begin{cases} \equiv R \text{ mod } n \Rightarrow \text{ valid signature} \\ \not\equiv R \text{ mod } n \Rightarrow \text{ invalid signature} \end{cases}$$
Security Considerations

The random value $R$ plays an essential role in the security of the scheme and its incorrect use can lead to the leak of the private key. If, by any chance, there is an instance where for two different signatures the private key $K_{pr}$ and the random number $R$ are identical, then the private key can be derived efficiently as follows.

Let us assume that two signatures $S_A$ and $S_B$, of the messages $m_1$ and $m_2$ respectively, have been created using the same private key $K_{pr}$ and the same random ephemeral key $K_E$, then

$$ S_A \equiv (h(m_1) + d \cdot R) \cdot K^{-1}_E \pmod{n} $$
$$ S_A \equiv (h(m_2) + d \cdot R) \cdot K^{-1}_E \pmod{n} \quad (68) $$

Then the random ephemeral key $K_E$ can be retrieved by

$$ K_E \equiv \frac{h(m_1) - h(m_2)}{S_A - S_B} \pmod{n} \quad (69) $$

Once the random ephemeral key $K_E$ is known, it can be used to retrieve the private key $K_{pr}$ as follows

$$ K_{pr} \equiv \frac{S_A \cdot K_E - h(m_1)}{R} \pmod{n} \quad (70) $$
References


