Bitcoin Developer Reference

Working Paper
Last changes: 30th July 2016

Krzysztof Okupski

Technische Universiteit Eindhoven, The Netherlands
`k.s.okupski@student.tue.nl`

Contents

1 Introduction

Bitcoin is a decentralized digital cryptocurrency created by pseudonymous developer Satoshi Nakamoto. The first paper on Bitcoin [?], also referred to as the original Bitcoin paper, was published by Nakamoto in 2008. It provides a brief description of the concepts and architecture schematics of the Bitcoin protocol. It was used as theoretical groundwork for the first implementation of a fully functional Bitcoin client. However, up until now, no structured and accessible protocol specification has been written. Although the Bitcoin community has successfully created a protocol specification [?], it requires solid prior understanding of its concepts and implementation. In this paper a formal and accessible specification of the core Bitcoin protocol, i.e. excluding the P2P overlay network, will be presented.

2 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.

2.1 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. 5.2) incorporates a proof of work system based on Adam Back's Hashcash [?]. 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 *double-SHA256* ($SHA256^2$) 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 $H = SHA256^2(msg||n)$ is less or equal to the target T is

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

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

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

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

$$SHA256^2(msg||n) \leq T \tag{3}$$

2.2 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. 2.1. Leaves are computed directly as hashes over data blocks, whereas nodes further up the tree are computed by concatenating and hashing their respective children.

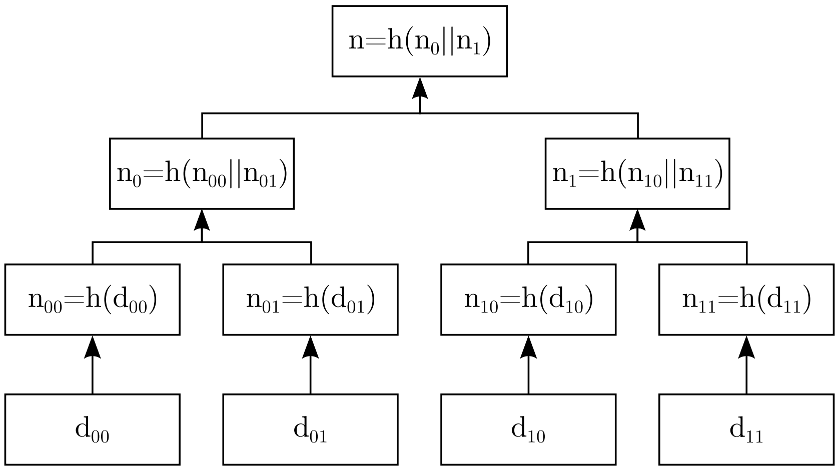


Figure 2.1. Merkle Tree

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.

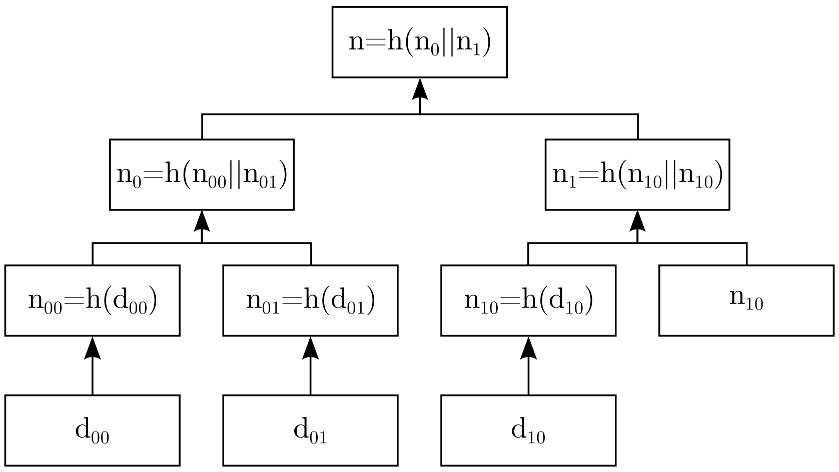


Figure 2.2. Merkle Tree with Missing Nodes

In the example given in Fig. 2.2 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. 2.1). Should an odd number of nodes occur at any other point during the computation, then the same rule is applied.

3 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. 5.2). In it the miner creates a *block* with a set of unprocessed transactions and attempts to solve a *proof of work puzzle* (see Sect. 2.1). 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 [?] and the Bitcoin Protocol Specification on Wikipedia [?]. Furthermore, all data types denoted in the diagrams are explained in detail in Appendix A.

3.1 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.

Field name	**Type (Size)**	**Description**
nVersion	int (4 bytes)	Block format version (currently 2).
HashPrevBlock	uint256 (32 bytes)	Hash of previous block header $SHA256^2(nVersion\|\| \dots \|\|nNonce)$.
HashMerkleRoot	uint256 (32 bytes)	Top hash of the Merkle tree built from all transactions.
nTime	unsigned int (4 bytes)	Timestamp in UNIX-format of approximate block creation time.
nBits	unsigned int (4 bytes)	Target T for the proof of work problem in compact format. Full target value is derived as: $T = 0xh_2h_3h_4h_5h_6h_7 * 2^{8*(0xh_0h_1-3)}$
nNonce	unsigned int (4 bytes)	Nonce allowing variations for solving the proof of work problem.
#vtx	VarInt (1-9 bytes)	Number of transaction entries in *vtx*.
vtx[ ]	Transaction (Variable)	Vector of transactions.

Table 3.1. Block Structure

nVersion
The version field stores the version number of the block format. Ever since BIP0034 [?] 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. 3.1.

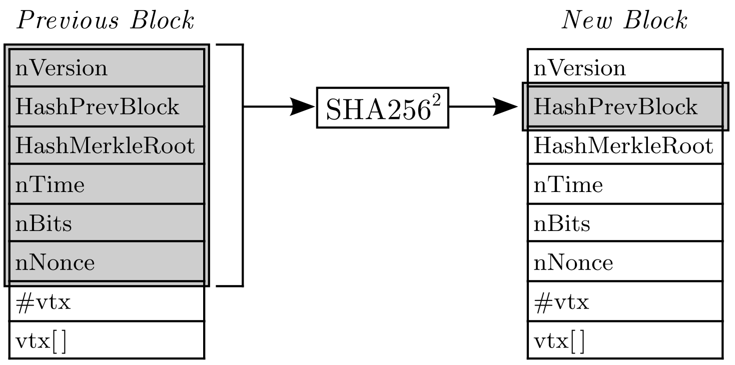


Figure 3.1. Block Reference Computation

A double-SHA256 hash is calculated over the concatenation of all elements in the previous block header:

$$SHA256^2(nVersion||HashPrevBlock||HashMerkleRoot||nTime||nBits||nNonce) \tag{4}$$

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. 2.2. 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. 5.2). 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

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

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 [?], the last 2015 blocks:

$$T' = \frac{t_{sum}}{14 * 24 * 60 * 60s} * T \tag{6}$$

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. 5.2.

3.2 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.

3.2.1 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*).

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

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:

$$TxID = SHA256^2(Transaction) \tag{7}$$

  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. 3.2.

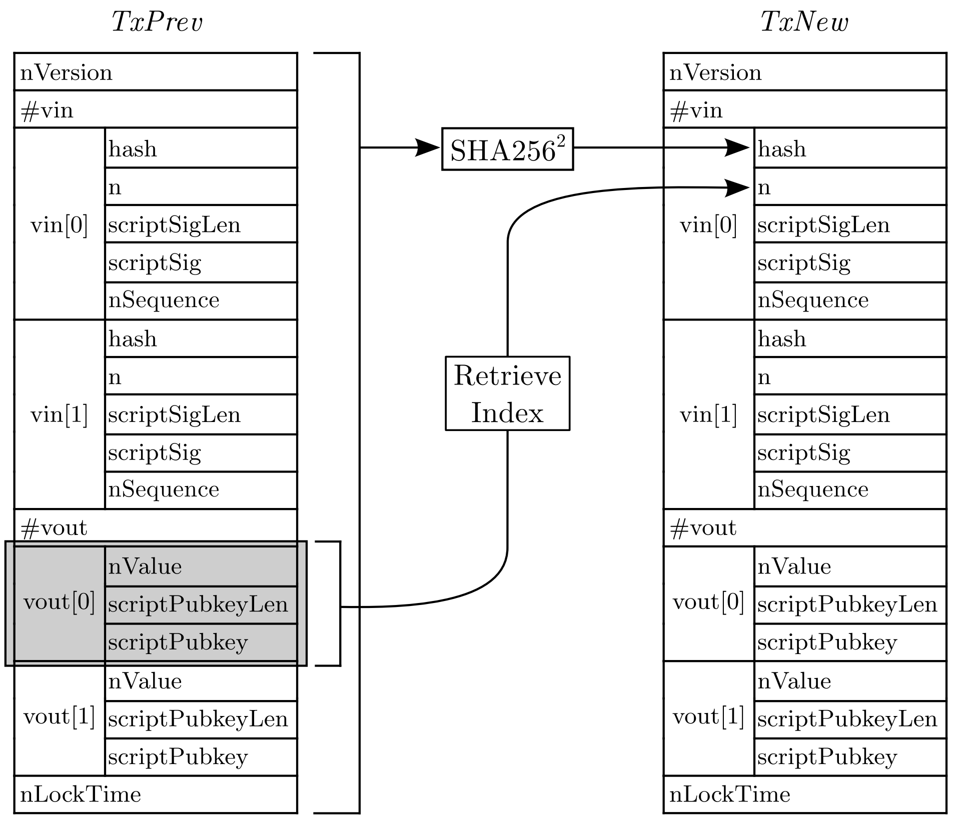


Figure 3.2. Transaction Output Reference Computation

- *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 *scriptPubey* 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. 4.2.
- *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. 3.3 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 [?]. 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. 4.2.

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:

Value	Description
0	Always locked.
$< 5 * 10^8$	Block number at which transaction is locked.
$\geq 5 * 10^8$	UNIX timestamp at which transaction is locked.

Table 3.3. Lock Time Values

If all transaction inputs (see *vin* field) have a final sequence number (see *nSequence* field), then the lock time is ignored. More details can be found in Sect. 3.3 under the Final Transaction Rule.

3.2.2 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.

Field name		**Type (Size)**	**Description**
nVersion		int (4 bytes)	Transaction format version (currently 1).
#vin		VarInt (1-9 bytes)	Number of transaction inputs entries in *vin*.
vin[ ]	hash	uint256 (32 bytes)	Fixed double-SHA256 hash.
	n	uint (4 bytes)	Fixed transaction output index.
	coinbaseLen	VarInt (1-9 bytes)	Length of *coinbase* field in bytes.
	coinbase	CScript (Variable)	Encodes the block height and arbitrary data.
	nSequence	uint (4 bytes)	Transaction input sequence number.
#vout		VarInt (1-9 bytes)	Number of transaction output entries in *vout*.
vout[ ]	nValue	int64_t (8 bytes)	Amount of 10^{-8} BTC.
	scriptPubkeyLen	VarInt (1-9 bytes)	Length of *scriptPubkey* field in bytes.
	scriptPubkey	CScript (Variable)	Script specifying conditions under which the transaction output can be claimed.
nLockTime		unsigned int (4 bytes)	Timestamp until which transactions can be replaced before block inclusion.

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:

$$\begin{aligned} hash &= 0 \\ n &= 2^{32} - 1 \end{aligned} \tag{8}$$

- *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.

Field name	**Size (bytes)**	**Description**
blockHeightLen	1	Length of *blockHeight* field in bytes.
blockHeight	*blockHeightLen*	Block height encoding.
arbitraryData	$coinbaseLen - (blockHeightLen+1)$	Arbitrary data field.

Table 3.5. Coinbase Field Encoding

As of BIP0034 [?], 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. 5.2).

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.

3.3 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* 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. 4.3).
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. 4.3) per transaction is permitted.

4 Bitcoin ownership

4.1 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. 4.1.

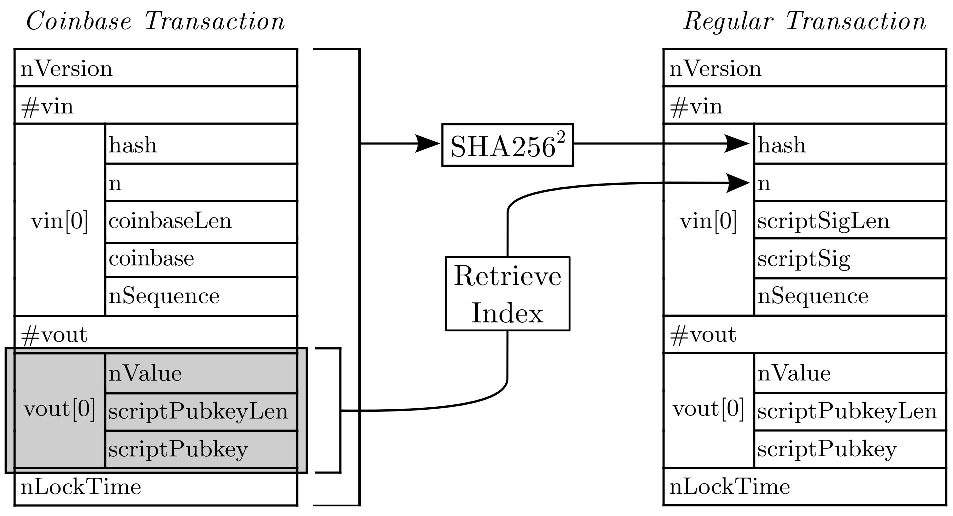


Figure 4.1. Bitcoin Transaction Chain

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. 4.5), 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.

4.2 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.

[2] See *https://en.bitcoin.it/wiki/Script* for complete reference.

4.3 Standard Transaction Types

4.3.1 Pay-to-Pubkey (P2PK)

The structure of the challenge and response scripts of a Pay-to-Pubkey transaction are depicted below in Fig. 4.2. 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. 4.2 `CHECKSIG` stands for signature verification.

```
scriptPubkey: <pubkey> OP_CHECKSIG
scriptSig:    <signature>
```

Figure 4.2. 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 4.1 and 4.2. The signature and the public key are pushed onto the stack and evaluated. Note that the signature is computed as described in Sect. 4.5.

Stack	**Remaining Script**	**Description**
`Empty`	`<signature>`	The signature is pushed on the stack.
`<signature>`	`Empty`	Final state after evaluating *scriptSig*.

Table 4.1. Pay-to-Pubkey *scriptSig* Execution

Stack	**Remaining Script**	**Description**
`<signature>`	`<pubkey> OP_CHECKSIG`	State after copying the stack of the signature script evaluation. The public key is pushed on the stack.
`<pubkey>` `<signature>`	`OP_CHECKSIG`	The signature is verified for the top two stack elements and the result is pushed on the stack.
`True`	`Empty`	Final state after evaluating *scriptPubkey*.

Table 4.2. Pay-to-Pubkey *scriptPubkey* Execution

4.3.2 Pay-to-PubkeyHash (P2PKH)

The structure of the challenge and response scripts of a Pay-to-PubkeyHash transaction can be seen below in Fig. 4.3.

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

Figure 4.3. Pay-to-PubkeyHash Structure

In a Pay-to-PubkeyHash transaction the sender transfers Bitcoins to the owner of a P2PKH address (see Sect. 4.4.1). He specifies in the challenge script the public key hash (*pubkeyHash*) of the Bitcoin address (depicted in Fig. 4.8) 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 4.3 and 4.4. 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. 4.5.

Stack	**Remaining Script**	**Description**
`Empty`	`<signature> <pubkey>`	The signature and the public key are pushed on the stack.
`<pubkey>` `<signature>`	`Empty`	Final state after evaluating *scriptSig*.

Table 4.3. Pay-to-PubkeyHash *scriptSig* Execution

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

Table 4.4. Pay-to-PubkeyHash *scriptPubkey* Execution

4.3.3 Pay-to-ScriptHash (P2SH)

The structure of the challenge and response scripts of a Pay-to-ScriptHash transaction is depicted below in Fig. 4.4.

```
scriptPubkey: OP_HASH160 <scriptHash> OP_EQUAL
scriptSig:    <signatures> {serializedScript}
```

Figure 4.4. Pay-to-ScriptHash Structure

In a Pay-to-ScriptHash transaction the sender transfers Bitcoins to the owner of a P2SH Bitcoin address (see Sect. 4.4.2). He specifies in the challenge script the serialized script hash (*scriptHash*) of the Bitcoin address (depicted in Fig. 4.9) and one requirement that the claimant has to prove:

1) Knowledge of the redemption script *serializedScript* corresponding to *scriptHash*.

To do so, the claimant creates a response script containing one or more signatures and the serialized redemption script *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 *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:

```
scriptPubkey: OP_HASH160 <scriptHash> OP_EQUAL
scriptSig:    <signatures> {<pubkey> OP_CHECKSIG}
```

Figure 4.5. 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 (*serializedScript*) is consistent with the redemption script hash (*scriptHash*) and then the transaction is evaluated using the redemption script as *scriptPubkey*. The evaluation is depicted in Table 4.5, 4.6 and 4.7.

Stack	**Remaining Script**	**Description**
`Empty`	`<signature>` `{<pubkey> OP_CHECKSIG}`	The signature and the redemption script are pushed on the stack.
`{<pubkey> OP_CHECKSIG}` `<signature>`	`Empty`	Final state after evaluating *scriptSig*.

Table 4.5. Pay-to-ScriptHash *scriptSig* Execution

Stack	**Remaining Script**	**Description**
`{<pubkey> OP_CHECKSIG}` `<signature>`	`OP_HASH160 <scriptHash>` `OP_EQUAL`	State after copying the stack of the signature script evaluation. The top stack element is first hashed with SHA256 and then with RIPEMD160.
`<scriptHashNew>` `<signature>`	`<scriptHash> OP_EQUAL`	The redemption script hash is pushed on the stack.
`<scriptHash>` `<scriptHashNew>` `<signature>`	`OP_EQUAL`	Equality of the top two stack elements is checked. The result of the evaluation is pushed on the stack.
`True` `<signature>`	`Empty`	Final state after evaluating *scriptPubkey*.

Table 4.6. Pay-to-ScriptHash *scriptPubkey* Execution

For the additional validation step the stack after *scriptSig* execution (see Table 4.5) 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 4.2).

Stack	**Remaining Script**	**Description**
`<signature>`	`<pubkey> OP_CHECKSIG`	Initial state of supplementary validation step.
...	...	...

Table 4.7. Pay-to-ScriptHash Supplementary Validation

4.3.4 Multisig

The structure of the challenge and response scripts of a Multisig transaction is depicted below in Fig. 4.6.

```
scriptPubkey: m <pubkey 1> ... <pubkey n> n OP_CHECKMULTISIG
scriptSig:    OP_0 <signature 1> ... <signature m>
```

Figure 4.6. 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 (*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 `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 `OP_0`. The script is then evaluated as depicted in Table 4.8 and 4.9. 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 *scriptSig* (500 bytes) and the allowed size of the serialized script *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 (*scriptSig*) will be increased in the next major release to 1650 bytes, thus allowing even bigger P2SH Multisig transactions.

Stack	Remaining Script	Description
`Empty`	`OP_0 <signature 1> ... <signature m>`	The signatures are pushed on the stack.

`<signature m>`
`...`
`<signature 1>`
`OP_0`
	`Empty`	Final state after evaluating *scriptSig*.

Table 4.8. Multisig *scriptSig* Execution

Stack	Remaining Script	Description

`<signature m>`
`...`
`<signature 1>`
`OP_0`
	`m <pubkey 1> ... <pubkey n> n OP_CHECKMULTISIG`	State after copying the stack of the signature script evaluation. The public keys are pushed on the stack.

`n`
`<pubkey n>`
`...`
`<pubkey 1>`
`m`
`<signature m>`
`...`
`<signature 1>`
`OP_0`
	`OP_CHECKMULTISIG`	The signatures are verified in order of appearance and the result is pushed on the stack.

`True`	`Empty`	Final state after evaluating *scriptPubkey*.

Table 4.9. Multisig *scriptPubkey* Execution

4.3.5 Nulldata

The structure of the challenge and response scripts of a Nulldata transaction is depicted below in Fig. 4.7.

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

Figure 4.7. 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.

4.4 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.

4.4.1 Pay-to-PubkeyHash Address

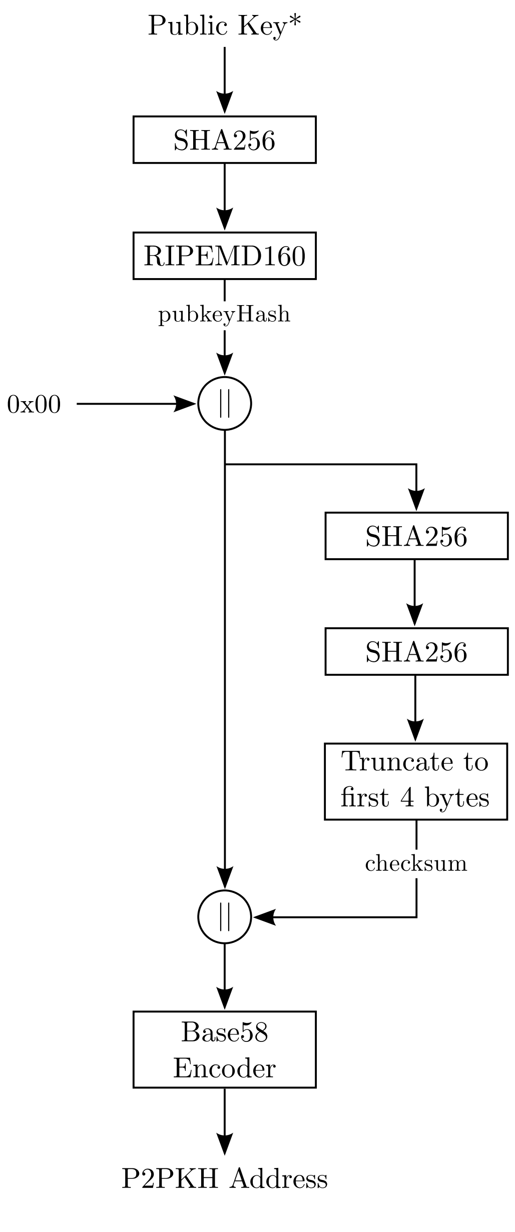
Essentially, a Pay-to-PubkeyHash address is a hash of the public key portion of the public-private ECDSA keypair with a built-in checksum. Schematics of how it is calculated can be seen in Fig. 4.8.

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 [?]. The final result is a *P2PKH address*.

4.4.2 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. 4.9.

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 [?]. The final result is a *P2SH address*.



*EC Public Key encoded as an uncompressed point on the secp256k1 curve according to the ANSI X9.62 standard

Figure 4.8. P2PKH Address Computation

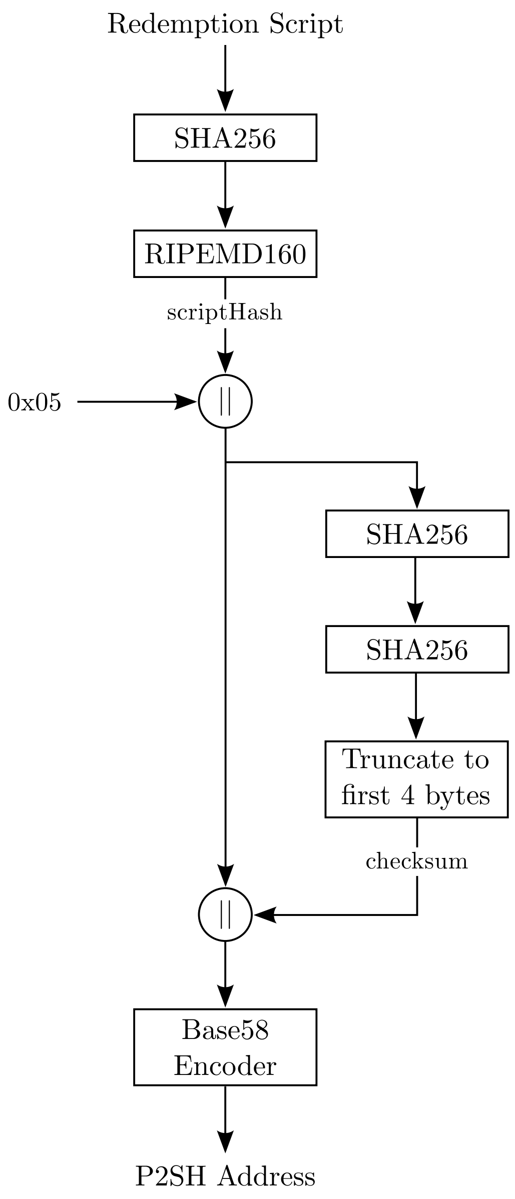


Figure 4.9. P2SH Address Computation

4.5 Signatures

Signatures are a central cryptographic primitive in Bitcoin and play a significant role in transaction authorization (see Sect. 4.1). 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.

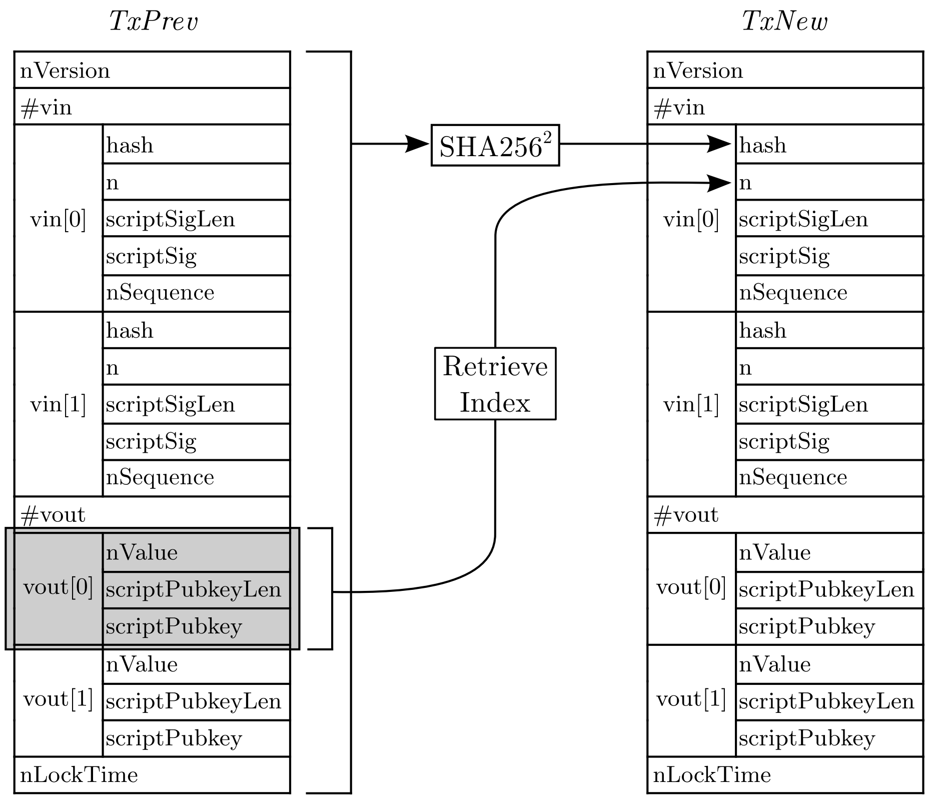


Figure 4.10. Signature Computation - Initial State

In the following the various signature types will be discussed in detail. The depiction in Fig. 4.10 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. 4.11 with grey fields.

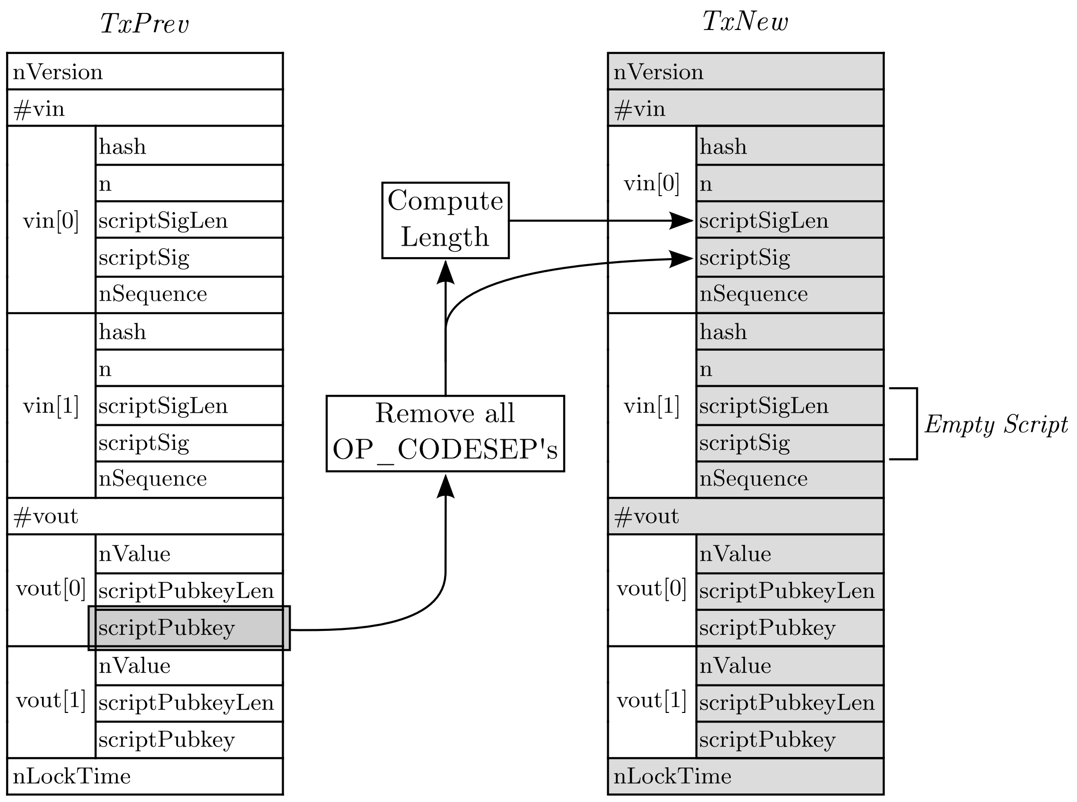


Figure 4.11. Signature Computation - SIGHASH_ALL

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 occurences 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. 4.12 with grey fields.

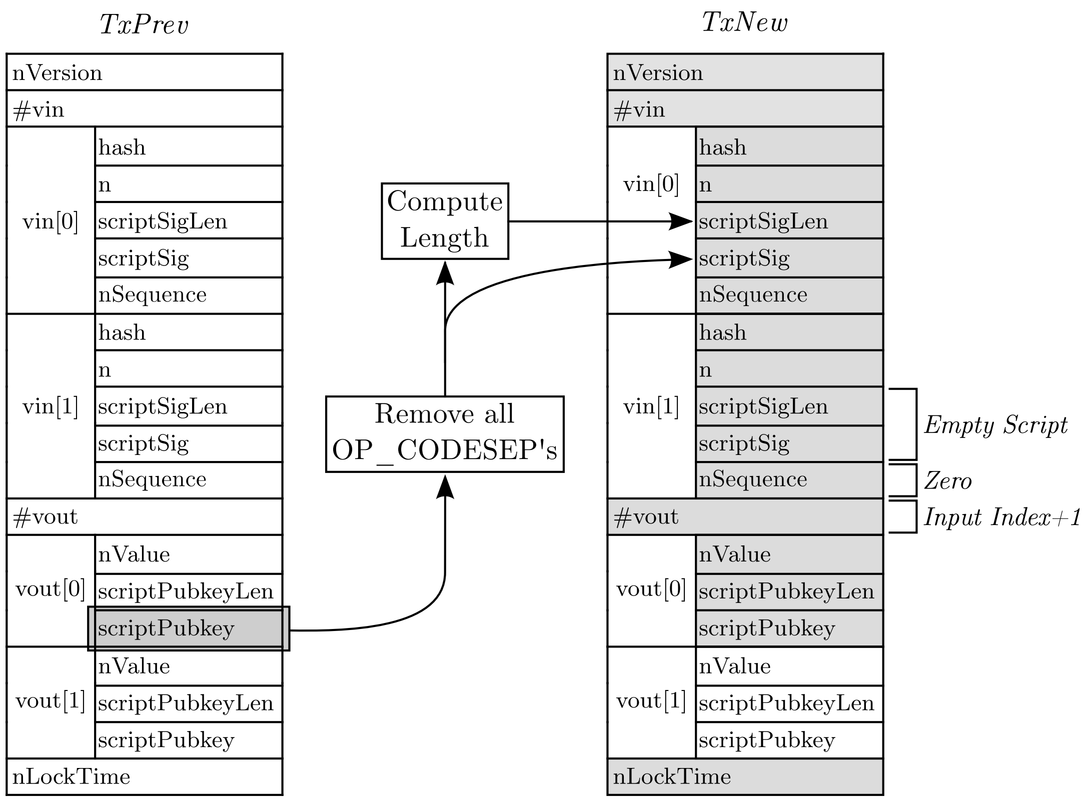


Figure 4.12. Signature Computation - SIGHASH_SINGLE

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 occurences 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.

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. 4.13 with grey fields.

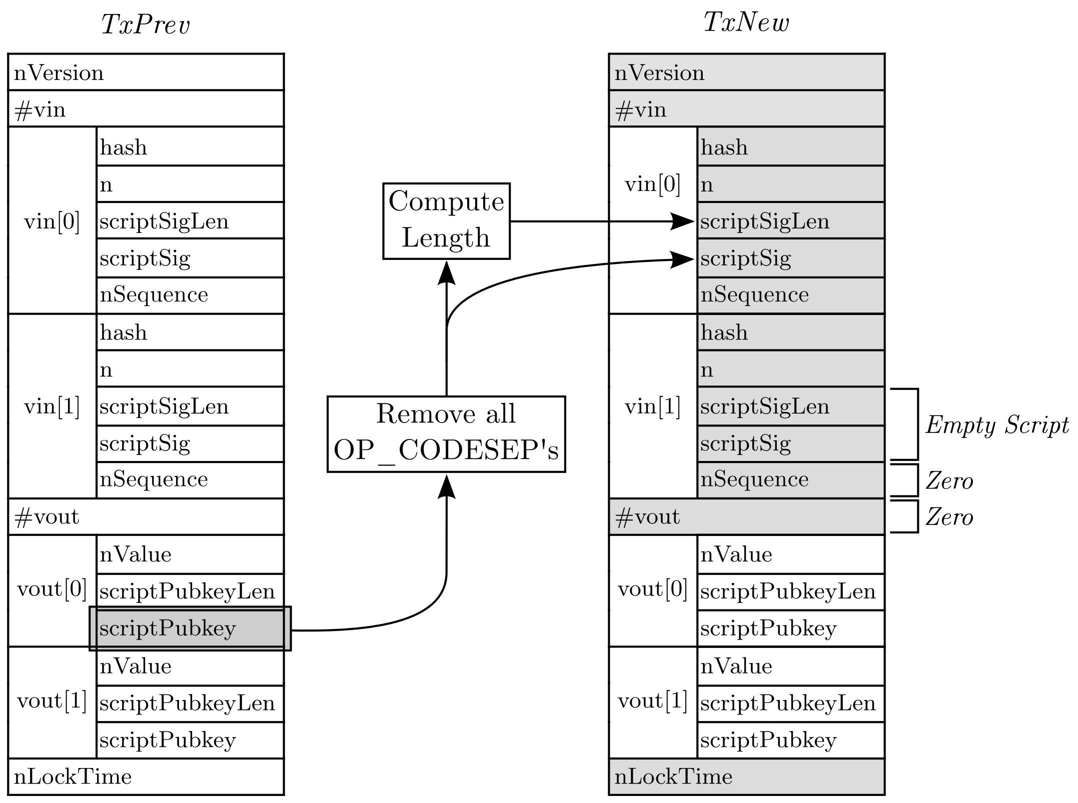


Figure 4.13. Signature Computation - SIGHASH_NONE

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 occurences 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. 4.14 illustrates that the second transaction input is excluded from the signature.

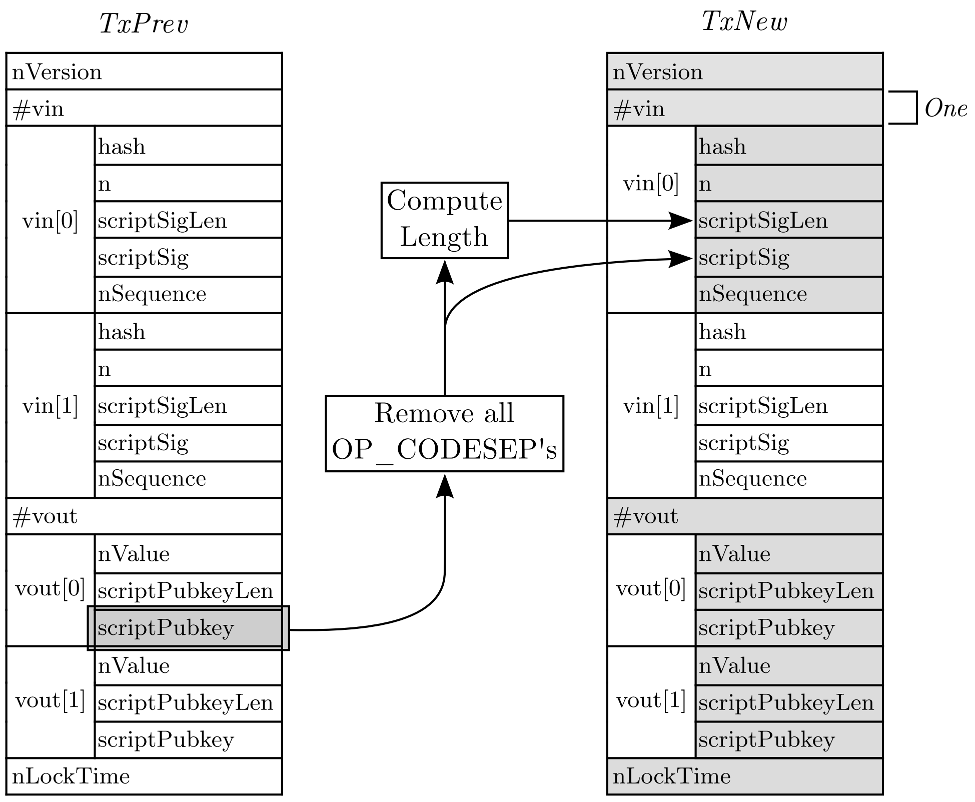


Figure 4.14. Signature Computation - SIGHASH_ALL|SIGHASH_ANYONECANPAY

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.

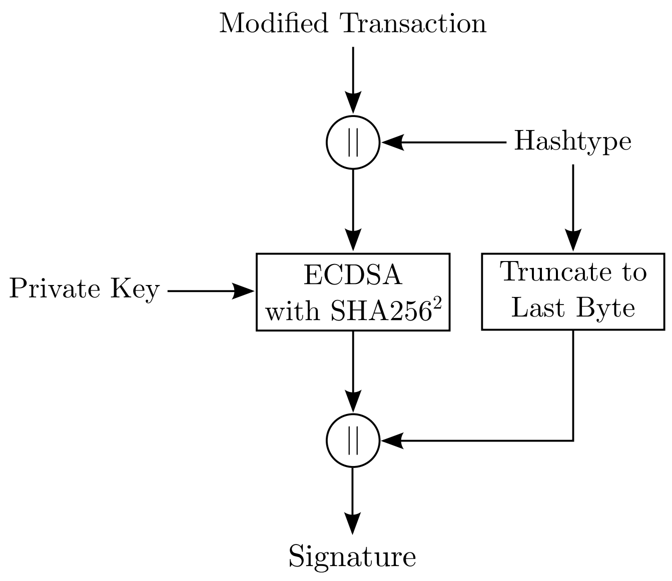


Figure 4.15. Signature Computation - Finalization

5 Blockchain

5.1 Structure

The blockchain is a record of all transactions that have occured 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. 3.1, 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.

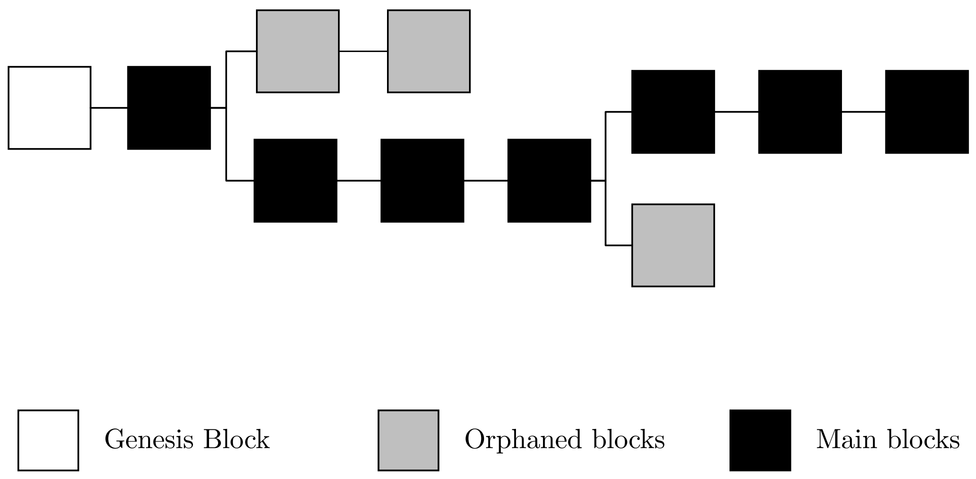


Figure 5.1. Blockchain

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. 5.1, 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.

5.2 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 [?], 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. 4.2 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 (*nNonce*) in the block header itself and the coinbase field (*coinbase*) in the coinbase transaction, which indirectly changes the Merkle root (*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

$$SHA256^2(nVersion||HashPrevBlock||HashMerkleRoot||nTime||nBits||nNonce) \tag{9}$$

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):

$$H \leq T \tag{10}$$

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.

Acknowledgements

I would like to thank Dr. Boris Škorić for his invaluable help in developing this paper, the Bitcoin Core developers Pieter Wuille (*sipa* on #bitcoin-dev), Luke-Jr (#bitcoin-dev) and everyone else who has provided me with their feedback.

Appendix A Data types

General data types

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

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.

Value interval	Size (Bytes)	Format
$[0, 2^8 - 3)$	1	uint8_t
$[2^8 - 3, 2^{16})$	3	0xFD followed by the value as uint16_t
$[2^{16}, 2^{32})$	5	0xFE followed by the value as uint32_t
$[2^{32}, 2^{64})$	9	0xFF followed by the value as uint64_t

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:

$$TxFee = TxFeeRate \cdot \left\lceil \frac{TxSize}{1000} \right\rceil \tag{11}$$

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

$$\frac{\frac{MinTxFeeRate}{1000} \cdot (TxOutSize + 148)}{nValue} > \frac{1}{3} \tag{12}$$

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

$$MinValue = 3 \cdot \frac{MinTxFeeRate}{1000} \cdot (TxOutSize + 148) \tag{13}$$

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.