Bitcoin Protocol Specification Protocol Version 0.8.6

Krzysztof Okupski

 $\label{thm:condition} Technische \ Universiteit \ Eindhoven, \ The \ Netherlands \\ \texttt{k.s.okupski@student.tue.nl}$

Table of Contents

1	Introduction				
2	Prel	iminaries			
	2.1	Proof of Work			
	2.2	Merkle Trees			
3	Arch	iitecture			
	3.1	Blocks			
	3.2	Transactions			
		Regular transactions			
		Coinbase Transactions			
4	Bitc	oin ownership			
	4.1	General			
	4.2	Bitcoin Addresses 16			
	4.3	Script			
	4.4	Signatures			
5	Bloc	kchain			
	5.1	Structure			
	5.2	Mining			
6	Ack	nowledgments			

1 Introduction

Bitcoin is a P2P digital cryptocurrency created by pseudonymous developer Satoshi Nakamoto. The first paper on Bitcoin [5], also referred to as the original Bitcoin paper, has been 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 present times, no structured and accessible protocol specification was created. Although the Bitcoin community has successfully created a protocol specification [8], it is meant as a reference for individuals with good prior understanding of the protocol. In the scope of 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 this paper. In particular, the proof-of-work protocol and Merkle Trees will be discussed. Note that Digital Signatures are required as well but are intentionally skipped since they are sufficiently covered by online literature.

2.1 Proof of Work

A proof-of-work is a cryptographic scheme used to ensure that a party has performed a certain amount of work. 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 proof-of-work 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. In turn, 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 $SHA256^2$ and the predefined structure 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

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

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

$$\frac{1}{\Pr[H \le T]} = \frac{2^{256}}{T}.$$
 (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:

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

2.2 Merkle Trees

Merkle trees, named after their creator Ralph Merkle, are binary hash trees used for efficient and secure verification of data integrity. However, note that Merkle trees alone do not guarantee integrity against active adversaries, as they can modify any data and re-compute the resulting tree at will. Nevertheless, under the assumption that the tree cannot be modified, any attempt of tampering with the data will be detected when the tree is verified.

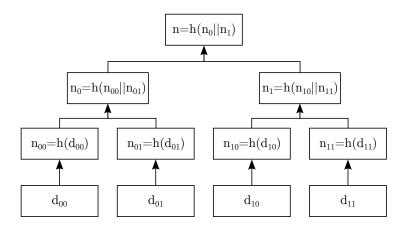


Fig. 2.1. Merkle Tree

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 their respective children.

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 and all nodes along the branch until the root. Therefore, the number of to be computed hashes scales logarithmically in the number of nodes. Since hashes are relatively small

in size, this process is fairly efficient. Finally, since the tree can be recreated from the supplied data blocks, it is sufficient to only store the root.

Corner cases

The case that is considered above is when the number of data blocks is a power of two. Naturally, this might not be the case. For this reason the situation has to be considered where the number of data blocks is not a power of two. Although there are several ways to deal with this, the solution used in the Bitcoin protocol will be presented.

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.

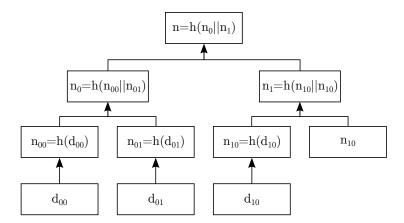


Fig. 2.2. Merkle Tree with odd node count

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 row is missing a child. The rest of the computation remains the same 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

Bitcoin is a decentralized digital currency based on a P2P network. Within the network each node is responsible for processing transactions and maintaining a public ledger of all transactions. Transactions are processed in a procedure called mining (see Sect. 5.2). The ledger, also known as the blockchain, is a record of all transactions that have ever occured in the Bitcoin system and is used to track ownership of Bitcoins (see Sect. 5). In this section the structure of blocks forming the blockchain will be discussed. Note that the following description is based on the Bitcoin source code [4] and the publicly available protocol specification [8]. Furthermore, all data types denoted in class diagrams are described 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 (HashPre-vBlock), the root of the Merkle tree (HashMerkleRoot), a timestamp (nTime), a target value (nBits) and a nonce (nNonce). Finally, the payload stores transactions 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 $SHA256^2(nVersion nNonce)$.
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[] CTransaction (Variable)		Vector of transactions.

Table 3.1. CBlock Class Description

nVersion

The version field stores the version number of the block format. Ever since BIP0034 [1] is in place, the block format version is 2. Furthermore, by now the

95% rule is in place, which states that all version 1 blocks should be rejected once 950 of the last 1000 blocks are version 2 or greater. Note that this change only affects miners.

HashPrevBlock

This field stores the reference to the previous block, computed as a hash over the block header as depicted in Fig. 3.1.

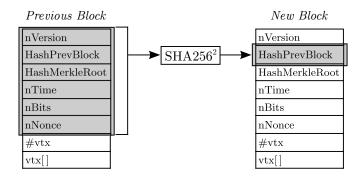


Fig. 3.1. Hash of Previous Block

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

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

The reference functions as a chaining link in the blockchain. By including a reference of 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 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 the block mining process, it is fixed at the beginning of the mining process.

nBits

The nBits field stores a compact representation of the target value T. The target value is a 256 hex-digit long number, whereas its corresponding compact representation is only 8 hex-digits long and thus encoded with only 4 bytes. The target value can be derived from its compact hexadecimal representation $0xh_0h_1h_2h_3h_4h_5h_6h_7$ with the formula:

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

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 growing network, the target T is recalculated every 2016 blocks based on the average time t_{avg} it took to mine one of these blocks:

$$T' = \frac{t_{avg}}{600s} * T \tag{6}$$

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 for circulation. They are included in every block as the very first transaction and are meant as a reward for solving a proof-of-work problem. Regular transactions, on the other hand, are used to transfer existing Bitcoins amongst different accounts. From an architectural point of view, the 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).

Field name		Type (Size)	Description	
nVersi	on	int (4 bytes)	Transaction format version (currently 1).	
#vin		VarInt (1-9 bytes)	Number of transaction inputs entries in <i>vin</i> .	
	hash	uint256 (32 bytes)	Double-SHA256 hash of a transaction.	
vin[]	n	uint (4 bytes)	Index of a transaction output within the transaction specified by <i>hash</i> .	
VIII[]	scriptSigLen	VarInt (1-9 bytes)	Length of scriptSig field in bytes.	
	scriptSig	CScript (Variable)	Script used to prove ownership of the transaction output $(hash,n)$.	
#vout		VarInt (1-9 bytes)	Number of transaction output entries in <i>vout</i> .	
	nValue	int64_t (8 bytes)	Amount of 10^{-8} BTC.	
vout[]	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 inclusion in block.	

Table 3.2. CTransaction Class Description (Regular Transactions)

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 digital signature script in bytes (scriptSigLen), the digital signature script (scriptSig) itself and a transaction sequence number (nSequence).

- (hash, n)

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

$$TxId = SHA256^{2}(Transaction)$$
 (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.

- scriptSiqLen

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 (*ScriptPubKey*) of the referenced transaction output (*prevout*). 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. 4.3.

- nSequence

This field stores the transaction sequence number. It is used for replacing transactions before they are included in a block. If the transaction is locked permanently, then the sequence number is set to the highest 4-byte integer OxFFFFFFFF.

One can first set the sequence number to 0 and replace the transaction later on by broadcasting another transaction with a higher sequence number. Note, however, that the transaction replacement mechanism is currently disabled.

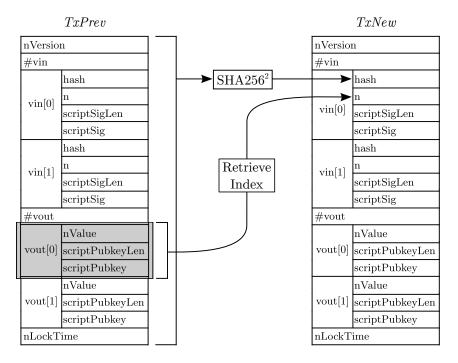


Fig. 3.2. Previous Output Reference

#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 being 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 transactions with outputs less than 5430 Satoshi, also called "dust", are considered non-standard [2]. Therefore, they will neither be relayed nor mined.

- 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 is used to prove that the transaction is allowed to claim it. More details on transaction verification can be found in Sect. 4.3.

nLockTime

This field stores the lock time of a transaction, that is a point in time at which the transaction should be included in a block. Once the lock time has been exceeded, the transaction is locked and becomes immune to transaction replacement. The lock time is encoded as either a timestamp in UNIX format or as a block number:

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

Table 3.3. Lock Time

If all transaction inputs (see vin field) have a final sequence number (0xFFFFFFFF), then this field is ignored. Note, however, that the transaction replacement mechanism is currently disabled.

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 for a coinbase transaction. In the following the differences between a regular and a coinbase transaction will be explained.

Field name		$egin{array}{c} { m Type} \ { m (Size)} \end{array}$	Description	
nVersion		int (4 bytes)	Transaction format version (currently 1).	
#vin		VarInt (1-9 bytes)	Number of transaction inputs entries in vin .	
	hash	$\begin{array}{c} \text{uint256} \\ \text{(32 bytes)} \end{array}$	Double-SHA256 hash of a transaction.	
vin[]	n	uint (4 bytes)	Index of a transaction output within the transaction specified by <i>hash</i> .	
VIII[]	coinbaseLen	VarInt (1-9 bytes)	Length of coinbase field in bytes.	
	coinbase	CScript (Variable)	Encodes the block height and arbitrary data.	
#vout		VarInt (1-9 bytes)	Number of transaction output entries in vout.	
	nValue	int64_t (8 bytes)	Amount of 10 ⁻⁸ BTC.	
vout[]	scriptPubkeyLen	VarInt (1-9 bytes)	Length of scriptPubkey field in bytes.	
	scriptPubkey	$ ext{CScript} \ ext{(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. CTransaction Class Description (Coinbase Transactions)

#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 reference to a previous output (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:

The hash does not reference any previous transaction output and is therefore set to zero whereas the output index is set to its maximal value $2^{32} - 1$.

- 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

The beginning of the field is reserved since block format version 2 for the block height. As described in BIP0034 [1], it is encoded in serialized CScript format, i.e. the first byte specifies the number of bytes used for encoding the block height, followed by the block height itself in little-endian notation. The remaining bytes can be chosen arbitrarily.

vout

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

- nValue

In a coinbase transaction the miner is allowed to transfer the current mining reward for solving the proof-of-work problem as well as all transaction fees. The reward 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 (referenced output values) and the sum of output values. The miner is allowed to claim both the mining reward and the transaction fee of all the included transactions.

4 Bitcoin ownership

4.1 General

Each user in the system maintains a wallet consisting of an arbitrary number of public-private keypairs. Keypairs are created at random on the secp256k1 Elliptic Curve [6]. Essentially, each keypair constitutes an account and transactions transfer the ownership of Bitcoins between them.

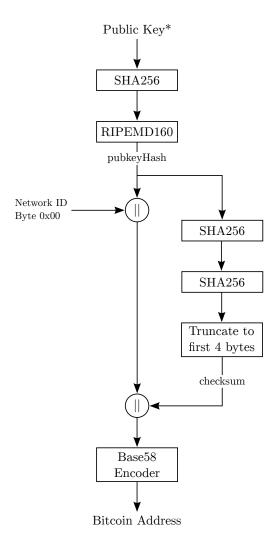
As described in Sect. 5.1, the blockchain maintans a public ledger of all transactions in the history of Bitcoin. It is used to infer the ownership of Bitcoins for various Bitcoin accounts in the system. From the creation of Bitcoins in a coinbase transaction through its entire lifetime, each transaction specifies to which Bitcoin address they belong. Note that in order to prevent double-spending attacks, coinbase transactions have a maturation time of 100 blocks and regular transactions have a confirmation time of 6 blocks.

Finally, the private key is used to create digital signatures, which is effectively a means to provide transaction authorization. Details on how this is done can be found in Sect. 4.3.

4.2 Bitcoin Addresses

A Bitcoin address is a unique identifier used as a destination for Bitcoin payments. It is between 27 and 34 alphanumeric characters long and is derived from a public key. In a simplified way, a Bitcoin address is a hash of the public portion of the public-private ECDSA keypair with a built-in checksum. Schematics of how the Bitcoin address is calculated can be seen in Fig. 4.1.

First, the EC public key is hashed using SHA256 and RIPEMD160. The resulting structure will be referred to as the public key hash pubkeyHash. Next, pubkeyHash is copied and a checksum is built by applying a double-SHA256 hash and truncating the result to the first 4 bytes. The network byte ID is prepended to pubkeyHash and the checksum is appended. Finally, the result is converted into a human-readable string using Base58 encoding [7]. The final result is the Bitcoin address.



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

Fig. 4.1. Bitcoin Address Computation

4.3 Script

Script is a stack-based, Turing-incomplete language designed specifically for the Bitcoin protocol. A script is essentially a set of instructions¹ 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*) accompanies a transaction output and specifies under which conditions it can be claimed.
- A response script (see *scriptSig*) accompanies a transaction input and is used to prove that the referenced output can be rightfully claimed.

Transactions are verified by first evaluating scriptSig, then copying the resulting stack and finally evaluating scriptPubKey of the referenced transaction output for all transaction inputs in it. 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 and only a restricted set of standard script templates is accepted. These are Pay-to-PubkeyHash, Pay-to-Pubkey, Pay-to-ScriptHash and Multisig. In the following, the structure of the two most common transaction types, Pay-to-PubkeyHash and Pay-to-Pubkey, will be discussed.

Pay-to-PubkeyHash

The structure of the challenge and response scripts in a transaction can be seen 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 DUP stands for duplication of the top stack element.

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

Fig. 4.2. Pay-to-PubkeyHash Structure

In a Pay-to-PubkeyHash transaction the sender transfers Bitcoins to the owner of a Bitcoin address. He specifies in the challenge script (*scriptPubkey*) the public key hash (*pubkeyHash*) of the Bitcoin address (depicted in Fig. 4.1) and two requirements that the claimant has to prove:

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

¹ See https://en.bitcoin.it/wiki/Script for complete reference.

To do so, the claimant creates a response script *scriptSig* with a signature and a public key. The script is then evaluated as depicted in Table 4.1 and 4.2. First, it verifies if the public key (*pubkey*) provided by the claimant corresponds to the public key hash (*pubkeyHash*) provided by the sender and then whether the signature is valid. The signature is computed as described in Sect. 4.4.

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

Table 4.1. Pay-to-PubkeyHash scriptSig Execution

Stack	Remaining Script	Description
<pre><pubkey> <signature></signature></pubkey></pre>	OP_DUP OP_HASH160 <pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG</pubkeyhash>	State after copying the stack of the signature script evaluation.
<puble> <puble> <puble> <puble> <pre> <pre< td=""><td>OP_HASH160 <pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG</pubkeyhash></td><td>The top stack element is duplicated.</td></pre<></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></puble></puble></puble></puble>	OP_HASH160 <pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG</pubkeyhash>	The top stack element is duplicated.
<pre><pubkeyhashx> <pubkey> <signature></signature></pubkey></pubkeyhashx></pre>	<pre><pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG</pubkeyhash></pre>	The top stack element is first hashed with SHA256 and then with RIPEMD160.
<pre><pubkeyhash> <pubkeyhashx> <pubkey> <signature></signature></pubkey></pubkeyhashx></pubkeyhash></pre>	OP_EQUALVERIFY OP_CHECKSIG	The public key hash is pushed on the stack.
<pre><puble>> <signature></signature></puble></pre>	OP_CHECKSIG	Equality of the top two stack elements is checked. If it evaluates to true then execution is continued. Otherwise it fails.
True	Empty	The signature is verified for the top two stack elements.

Table 4.2. Pay-to-Pubkey Hash scriptPubkey Execution

Pay-to-Pubkey

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

scriptPubkey: <pubkey> OP_CHECKSIG
scriptSig: <signature>

Fig. 4.3. 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 with only a signature. The script is then evaluated as depicted in Table 4.3 and 4.4. The signature and the public key are pushed onto the stack and evaluated. Note that the signature is computed as described in Sect. 4.4.

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

Table 4.3. Pay-to-Pubkey scriptSig Execution

Stack	Remaining Script	Description
<signature></signature>	<pre><pubkey> OP_CHECKSIG</pubkey></pre>	State after copying the stack of the signature script evaluation.
<pre><pubkey> <signature></signature></pubkey></pre>	OP_CHECKSIG	The signature is pushed on the stack.
True	Empty	The signature is verified for the top two stack elements.

Table 4.4. Pay-to-Pubkey scriptPubkey Execution

4.4 Signatures

Signatures are a central cryptographic primitive in the Bitcoin protocol used to prove ownership of the private key corresponding to a public key or a Bitcoin address. For a transaction a signature is included in each of the transaction input scripts scriptSig to prove that the referenced transaction outputs are indeed owned by the claimant.

Whenever the claimant computes a signature for a transaction input, he specifies one of four signature types. The various signature types are listen in Table 4.5. Depending on his choice different parts of the transaction will be covered by the signature.

Name	Value
SIGHASH_ALL	0x0000001
SIGHASH_NONE	0x00000002
	0x0000003
SIGHASH_ANYONECANPAY	0x00000080

Table 4.5. Hashtype Values

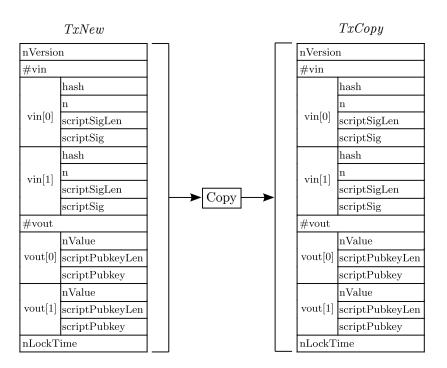
The signing process can be split into two phases - a preparation phase and a finalization phase. Whilst the preparation phase is constant, the finalization phase depends on the type of signature that is applied. For the sake of brevity, only the preparation phase and the default signature type SIGHASH_ALL will be explained.

Preparation

The process for computing a signature is not completely straightforward. Before a signature is computed, the transaction that is to be signed undergoes several modifications. Figure 3.2 depicts an example of a transaction with two inputs and two outputs.

Step 1

In the first step a transaction copy TxCopy of the transaction TxNew containing the to be signed transaction input is created.



 ${\bf Fig.\,4.4.}$ Signature Preparation - First Step

Step 2

In the second step all transaction input scripts in TxCopy are removed and the signature script lengths (scriptSigLen) are set to 0x00.

Step 3

Next, the public key script (scriptPubkey) of the referenced transaction output is taken, all occurences of $\mathtt{OP_CODESEPARATOR}$ are removed and the result is inserted into the signature script field (scriptSig) of the to be signed input. Finally, the signature script length (scriptSigLen) is updated to the length of the inserted script.

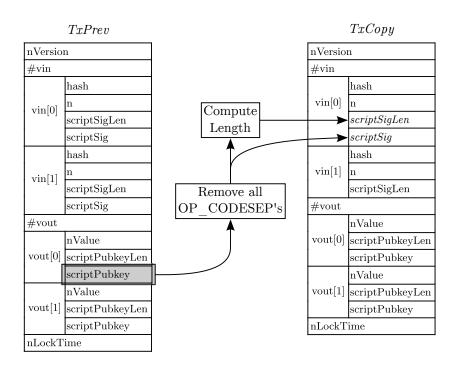


Fig. 4.5. Signature Preparation - Second and Third Step

Hashtype SIGHASH_ALL

In the default case no further modifications are made to the transaction copy TxCopy. The rest is computed as depicted in Fig. 4.6. Firstly, a 4-byte hashtype (see Table 4.5) in little-endian notation is appended to TxCopy. Next, an ECDSA signature, with double-SHA256 and the secp256k1 elliptic curve as parameters, is created. The intermediary result is here referred to as sig. Finally, a signature in Bitcoin constitutes of the concatenation of sig and an appended hashtype hType truncated to the last byte. It is necessary to append the hashtype (hType) to the intermediary signature sig to signal verifying parties what signature type was applied by the signing party.

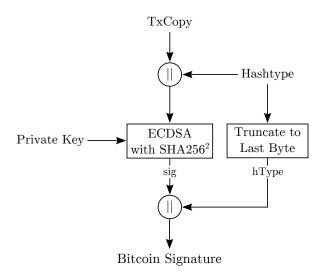


Fig. 4.6. Signature Finalization - SIGHASH_ALL

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. It is used to infer how many Bitcoins are owned by an address (or public key) in the system at any point in time. The novelty of Bitcoin lies, among other things, in how the blockchain is structured in order to guarantee a 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 enforces a chronological order on the blockchain, since it is not possible to create a valid hash of the previous block header without its prior 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. Nodes collectively 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 until the newest block. In order to achieve that, the adversary would have to computationally outperform the rest of the network, which is considered infeasible.

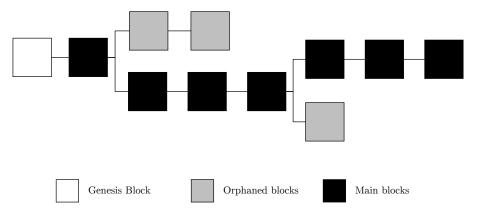


Fig. 5.1. Blockchain

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 mine are called miners. As described in [3], mining nodes perform the following steps in an endless loop:

- 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 what to do with it.
- 2) Verify all transactions that are to be included in the block. Transactions are verified as described in Sect. 4.3 and it is checked whether they have been previously spent.
- 3) Select the most recent block on the longest path in the blockchain, that is the path that involves most accumulated computational effort, and insert the hash of the block header into the new block.
- 4) Attempt to solve the proof-of-work problem as described below and broad-cast the solution. Should another node solve the proof-of-work problem before, then the block is first validated 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 choosing.

2) Compute the hash of the block header as

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

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

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

This process is repeated for various values of nNonce and coinbase 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.

6 Acknowledgments

I would like to thank Dr. Boris Škorić for his invaluable help in developing this paper and the Bitcoin developer Pieter Wuille, also known as *sipa* from #bitcoin-dev for, among other things, his input on Script.

References

- Gavin Andresen. Bitcoin Improvement Proposal 0034. https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki. Online; Accessed on 3.3.2014 at 17:52.
- 2. Gavin Andresen. Treat dust outputs as non-standard. https://github.com/bitcoin/bitcoin/pull/2577. Online; Accessed on 4.3.2014 at 20:19.
- 3. Chris Clark. Bitcoin Internals A technical guide to Bitcoin, 2013.
- 4. Satoshi Nakamoto et al. Bitcoin Source Code. https://github.com/bitcoin/bitcoin. Online; Accessed on 23.12.2013 at 18:22.
- 5. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- 6. Certicom Research. Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. http://www.secg.org/collateral/sec2_final.pdf. Online; Accessed on 16.2.2014 at 22:30.
- 7. Unknown. Base58Check Encoding. https://en.bitcoin.it/wiki/Base58Check_encoding. Online; Accessed on 26.3.2014 at 22:32.
- 8. Unknown. Bitcoin Protocol Specification. https://en.bitcoin.it/wiki/Protocol_specification. Online; Accessed on 11.2.2014 at 14:20.

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.
umitiou	20	Used for storing RIPEMD160 hashes as a byte array.
uint256	32	Unsigned integer array uint32_t[] of size 8.
umtzoo		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 integes is depicted below.

Value	Size (bytes)	Format
< 0xFD	1	uint8_t
= 0xFFFF	3	0xFD followed by the value as uint16_t
<= 0xFFFFFFF	5	0xFE followed by the value as uint32_t
<= 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	9	0xFF followed by the value as uint64_t