An Introduction to Bitcoin

Saravanan Vijayakumaran
Department of Electrical Engineering
Indian Institute of Technology Bombay
Email: sarva@ee.iitb.ac.in

Version 0.1
October 4, 2017

Abstract

Lecture notes on Bitcoin prepared for a Autumn 2018 course on cryptocurrencies at IIT Bombay.

Contents

1 Introduction **1**

2 Elliptic Curve Cryptography **3**
2.1 Groups 5
2.2 Fields 10
2.3 Elliptic Curves over Real Numbers 13
2.4 Elliptic Curves over Finite Fields 18
2.5 ECDSA 20

3 Cryptographic Hash Functions **24**
3.1 SHA-256 25
3.2 RIPEMD-160 31
3.3 P2PKH Addresses 32

4 The Blockchain **38**
4.1 Rewarding Blockchain Updation 39
4.2 The Block Header 40
4.3 Mining 43
4.4 Bitcoin Transactions 50
4.5 Bitcoin Ownership 54
4.6 Double Spending Attacks 55
4.7 Blockchain Integrity 59
4.8 The 51% Attacker 61
4.9 Summary 62

5 Bitcoin Transactions **63**
5.1 Block Format 63
5.2 Pre-SegWit Regular Transactions 65
5.3 Pre-SegWit Coinbase Transactions 71
5.4 Bitcoin Script 74
5.5 Pre-SegWit Standard Scripts 79
5.6 Pre-SegWit Signature Generation 96
5.7 Transaction Malleability 105
5.8 SegWit Standard Scripts 109

5.9 SegWit Regular and Coinbase Transactions 114
5.10 SegWit Signature Generation 118
5.11 Block Size and Sigop Limits 122

6 Contracts **127**
6.1 Escrow 127
6.2 Micropayments 128
6.3 Decentralized Lotteries 132

7 Bitcoin Development **143**
7.1 Bitcoin Improvement Proposals 143
7.2 Hard and Soft Forks 145

A ECDSA Signature Malleability **148**

B Probability of a successful double spending attack **150**

Chapter 1

Introduction

Bitcoin is a decentralized cryptocurrency. But what is a cryptocurrency? While any currency is a system for storing and transferring value, a cryptocurrency has the additional property that the notion of ownership of its units is established using cryptography. In the Bitcoin system, the transfer of bitcoins[1] between entities requires the sender to provide a digital signature proving ownership of the bitcoins being transferred. So what about the decentralized part? Bitcoin is a decentralized system because the creation of new bitcoins and the recording of all bitcoin transfers (which are called transactions) is performed by a peer-to-peer (P2P) network. Anyone can join the Bitcoin network by running open-source software freely available on the Internet.

The Bitcoin system provides the infrastructure necessary for enabling transactions between entities. This infrastructure consists of the following:

- A system for generating addresses where bitcoins can be received and stored.
- A method for ensuring that only the rightful owner of bitcoins stored in an address can move them to a new address.
- A database of all past transactions which is used to prevent double spending of the bitcoins stored in an address. This database is called the *blockchain*.

The traditional banking system also provides such an infrastructure. We have bank accounts where we can receive and store money. We can transfer money from one bank account to another using a cheque or an online account password. As each withdrawal from an account is stored in the bank's database, there is no possibility of double spending the money in an account. The surprising thing about Bitcoin is that this infrastructure is provided by a P2P

[1] It is convention to use Bitcoin (the word beginning with an uppercase B) to denote the cryptocurrency system and bitcoin to denote the unit of the cryptocurrency.

network where the participants are anonymous and not accountable for their behaviour.

The main innovation in Bitcoin is that the maintenance of the blockchain database is linked to the creation of new bitcoins. The blockchain consists of a linked list or chain of *blocks* where each block contains a set of transactions. Blocks are appended to the blockchain one at a time where each addition requires finding a solution to a computationally hard search problem. Nodes in the Bitcoin network which successfully add a block to the blockchain are rewarded with new bitcoins. Such nodes are called *miners* and their search for solutions of the computationally hard problems is called *mining*.

In the forthcoming chapters, we will describe the different aspects of the Bitcoin system in detail.

Chapter 2

Elliptic Curve Cryptography

In public key cryptography, each entity owns a pair of related keys: a public key and a private key. Given the private key, it is easy to calculate the corresponding public key. Finding the private key from the public key is computationally hard. As the names suggest, the private key needs to be kept a secret while the public key can be advertised. When public key cryptography is used for encrypted message transmission, the sender uses the receiver's public key to encrypt the message. The receiver can decrypt the encrypted message using its private key. It is computationally infeasible to decrypt the encrypted message without knowledge of the private key. When public key cryptography is used for implementing digital signatures, the signer uses its private key to create the signature on a given message. Verifying the validity of a signature on a message only requires the public key of the signer. It is computationally infeasible to create a valid signature without knowledge of the private key. These concepts are illustrated in Figure 2.1.

Elliptic curve cryptography (ECC) is a method for implementing public key cryptography. Bitcoin uses public keys derived from the `secp256k1` elliptic curve[1] to derive Bitcoin addresses. Ownership of a Bitcoin address is proved by generating a digital signature using the corresponding private keys and the elliptic curve digital signature algorithm (ECDSA). Such a proof of ownership is required in order to spend the bitcoins which have been received by a Bitcoin address.

Understanding the specifics of the ECC used in Bitcoin requires knowledge of some abstract algebra. In order to motivate the required prerequisites, let us look at the structure of the private and public keys as specified by the `secp256k1` domain parameters. All undefined terms will be discussed in the following sections.

[1]The `secp256k1` elliptic curve domain parameters are specified in `http://www.secg.org/sec2-v2.pdf`.

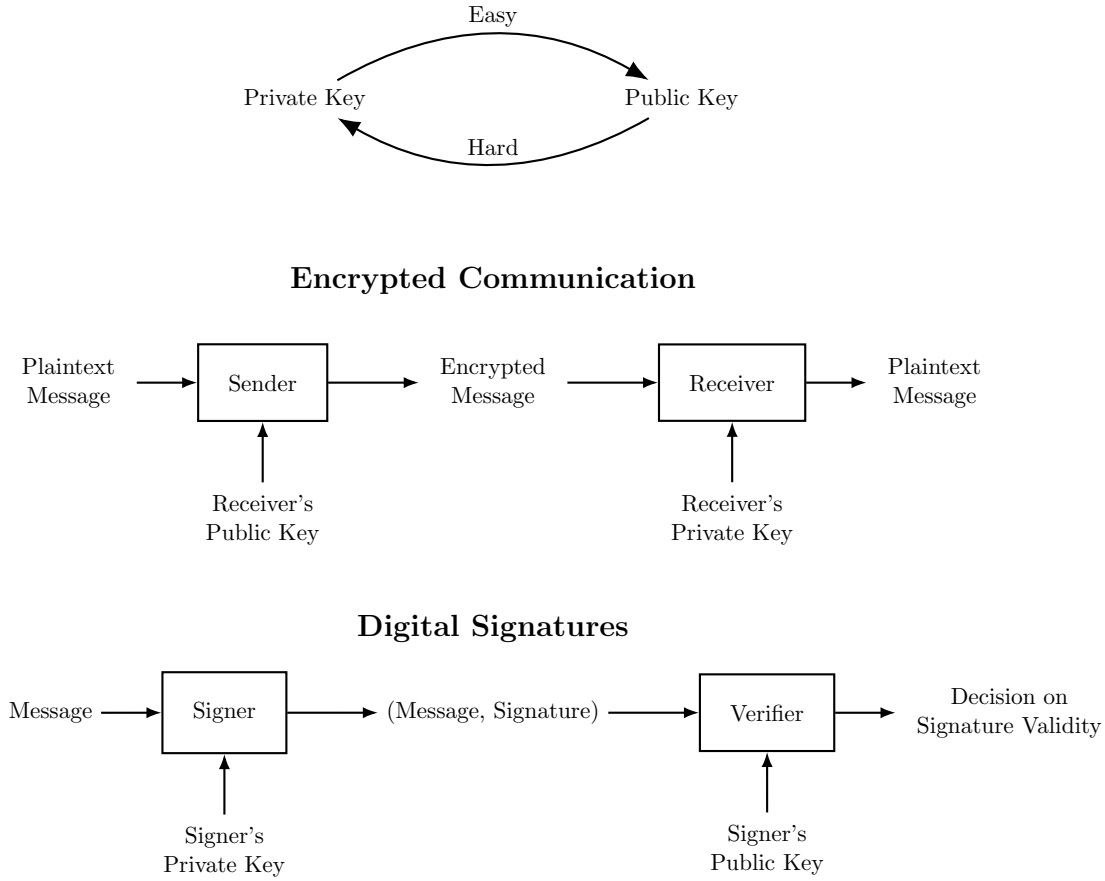


Figure 2.1: Public Key Cryptography Concepts

Let p be a 256-bit prime number given in hexadecimal format as

$$p = \underbrace{\texttt{FFFFFFFF FFFFFFFF} \cdots \texttt{FFFFFFFF}}_{\text{48 hexadecimal digits}} \texttt{FFFFFFFE FFFFFC2F}. \tag{2.1}$$

Let $\mathbb{F}_p$ denote the corresponding finite field. Consider the solutions $(x, y) \in \mathbb{F}_p^2$ to the equation

$$y^2 = x^3 + 7. \tag{2.2}$$

The solutions are nothing but points on the curve with coordinates from $\mathbb{F}_p$. Let E be the set of such points. If we add a special element $\mathcal{O}$ called the "point at infinity" to E, then a binary operation can be defined on $E \cup \{\mathcal{O}\}$ which makes $E \cup \{\mathcal{O}\}$ into a group. This binary operation is called "addition" for convenience and denoted by $+$. The set $E \cup \{\mathcal{O}\}$ is called the elliptic curve corresponding to $y^2 = x^3 + 7$ over $\mathbb{F}_p$.

The cardinality or order of the group $E \cup \{\mathcal{O}\}$ is a 256-bit prime number given by

$$n = \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141.} \tag{2.3}$$

A private key is simply a 256-bit integer k in the range $\{1, 2, \ldots, n-1\}$. The `secp256k1` domain parameter specification includes a base point $P \in E$ whose coordinates are given by

$$\begin{aligned} x &= \texttt{79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,} \\ y &= \texttt{483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.} \end{aligned} \tag{2.4}$$

The public key corresponding to the private key k is given by

$$\underbrace{P + P + \cdots + P + P}_{k \text{ times}} \tag{2.5}$$

where $+$ is the binary operation defined on $E \cup \{\mathcal{O}\}$. It is convenient to denote the public key as kP. The security of elliptic curve cryptography is based on the fact that the best known algorithms for finding the private key k given the public key kP have running time $O\left(\sqrt{n}\right)$. For the `secp256k1` domain parameters, the running time is $O\left(2^{128}\right)$.

The above discussion has hopefully motivated the need for studying groups, finite fields, and elliptic curves over finite fields. We will define these algebraic objects and describe some of their properties that are relevant to Bitcoin. We will then describe the ECDSA. While it is possible to consider the ECDSA as a black box which can create digital signatures, such a high-level treatment prevents one from understanding subtle issues like the non-uniqueness of digital signatures.

2.1 Groups

Let G be a set. A binary operation on G is a rule for combining pairs of elements from G. Familiar examples of binary operations are addition and multiplication over the integers.

Definition 1 (Group)**.** *Let G be a set with a binary operation $*$ defined on it. G is called a group if it satisfies the following properties:*

(i) $*$ is closed: For all $x, y \in G$, $x * y$ belongs to G.

(ii) $*$ is associative: For all $x, y, z \in G$,

$$(x * y) * z = x * (y * z).$$

(iii) Identity exists: There exists an element $e \in G$ such that

$$x * e = e * x = x$$

for all $x \in G$. This element is called the identity of the group.

(iv) Inverses exist: For every $x \in G$, there exists an element $y \in G$ such that

$$x * y = y * x = e$$

where e is the identity element. The element y is called the inverse of x and is denoted by x^{-1} or $-x$.

Some comments are in order before we look at examples of groups.

- A set G can have more than one binary operation defined on it. While the above definition refers to the set G as a group, the binary operation which endows G with the group structure needs to be specified explicitly in case of ambiguity.

- The parentheses in the associativity condition on $*$ indicate the order in which the operations are carried out. The expression $(x * y) * z$ indicates that $x * y$ is calculated first and the result is combined with z. The associativity property implies that the order of operations is irrelevant and the parentheses can be disregarded. We can use $x * y * z$ to represent both $(x * y) * z$ and $x * (y * z)$. Not all binary operations are associative. For example, consider subtraction over the integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Let $x = 1, y = 2, z = 3$. Then $(x - y) - z = -4 \neq 2 = x - (y - z)$.

- The element e is called "the" identity element of the group because it is unique. To see this, suppose there were two elements $e, e' \in G$ which satisfy the requirements of being an identity. Then $x = x * e$ for all $x \in G$ and $e' * y = y$ for all $y \in G$. Setting $x = e'$ and $y = e$ gives us $e' = e' * e = e$.

Here are some examples of groups.

- The set of integers $\mathbb{Z}$ with addition as the binary operation is a group. Addition of integers is closed and associative. Zero is the identity element of this group and the inverse of $x \in \mathbb{Z}$ is $-x$.

- The set of non-zero real numbers $\mathbb{R} \setminus \{0\}$ with multiplication as the binary operation is a group. We need to exclude 0 to get a group since the multiplicative inverse of 0 does not exist in $\mathbb{R}$. Multiplication of real numbers is closed and associative. The number 1 is the identity and the inverse of $x \in \mathbb{R} \setminus \{0\}$ is $\frac{1}{x}$.

- Consider the subset of the integers $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ where n is a positive integer. Let $+_n$ denote integer addition modulo n, i.e. $a +_n b = a + b \bmod n$ for $a, b \in \mathbb{Z}_n$. It is closed and associative. The inverse of 0 is 0 itself and the inverse of $x \in \mathbb{Z}_n \setminus \{0\}$ is $n - x$. Thus $\mathbb{Z}_n$ with $+_n$ as the binary operation is a group with 0 as the identity element.

Definition 2 (Group Order)**.** *The cardinality of a group G is called the order of G. Groups with finite order are called finite groups.*

In the examples we considered $\mathbb{Z}$ and $\mathbb{R} \setminus \{0\}$ are infinite groups, while $\mathbb{Z}_n$ is a finite group. In Bitcoin, we will encounter only finite groups. We will discuss some properties of finite groups which will help us better understand the structure of ECDSA with `secp256k1` domain parameters.

Subsets of a group can themselves have a group structure under the same operation. Such subsets are called subgroups.

Definition 3 (Subgroup)**.** *Let G be a group with $*$ as the binary operation. A subset H of G is called a subgroup of G if H is a group under the same operation $*$.*

Here are some examples of subgroups.

- G is a subgroup of itself.
- Let e be the identity of a group G. Then the singleton set $\{e\}$ is a subgroup of G.
- Let $\mathbb{Z}_e = \{2x \mid x \in \mathbb{Z}\}$ be the set of even integers. Then $\mathbb{Z}_e$ is a subgroup of $\mathbb{Z}$ with addition as the binary operation.
- The set of non-zero rational numbers $\mathbb{Q} \setminus \{0\}$ is a subgroup of $\mathbb{R} \setminus \{0\}$ under multiplication.
- Let $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ be a group with integer addition modulo 6 as the operation. Then $H = \{0, 3\}$ is a subgroup of $\mathbb{Z}_6$.

In the last example, the order of H is 2 which divides the order 6 of $\mathbb{Z}_6$. This is not a coincidence. It is an example of Lagrange's theorem for finite groups which we state without proof.

Theorem 1 (Lagrange's theorem)**.** *Let H be a subgroup of a finite group G. Then the order of H divides the order of G.*

Let us return to $\mathbb{Z}_6$ to motivate the next result. Note that the subgroup $H = \{0, 3\}$ can be written as $\{3 + 3, 3\}$. Now consider an arbitrary element in $\mathbb{Z}_6$, say 4. By repeatedly adding 4 to itself modulo 6 we get $\{4, 4 + 4 = 2, 4 + 4 + 4 = 0\} = \{0, 2, 4\}$, which is also a subgroup of $\mathbb{Z}_6$. Again, this is not a coincidence but an example of the following theorem about finite groups.

Theorem 2. *Let G be a finite group with operation $*$. Let $x \in G$ and x^n denote*

$$\underbrace{x * x * \cdots * x * x}_{n \text{ times}}.$$

Then $\langle x \rangle = \{x^n \mid n = 1, 2, 3, \ldots\}$ is a subgroup of G for any $x \in G$. It is called the subgroup generated by x.

Proof. Since $*$ is closed on G, $\langle x \rangle$ is a subset of G. To prove that $\langle x \rangle$ is a subgroup of G, we have to prove four properties.

(i) Closure: For all $u, v \in \langle x \rangle$, $u * v \in \langle x \rangle$.

Let $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$ be the set of positive integers. To prove closure, let $u = x^m$ and $v = x^n$ where $m, n \in \mathbb{Z}^+$. Then

$$u * v = x^m * x^n = (\underbrace{x * \cdots * x}_{m \text{ times}}) * (\underbrace{x * \cdots * x}_{n \text{ times}}) = \underbrace{x * \cdots * x}_{m+n \text{ times}},$$

where the last equality follows from the fact that x^m and x^n are also elements of the group G where the associativity of $*$ allows us to disregard the parentheses.

(ii) Associativity: For all $u, v, w \in \langle x \rangle$, $(u * v) * w = u * (v * w)$.

Since $(u * v) * w = u * (v * w)$ for all $u, v, w \in G$ and $\langle x \rangle \subseteq G$, the associativity of $*$ holds in $\langle x \rangle$.

(iii) Identity: There exists $e \in \langle x \rangle$ such that $e * u = u * e$ for all $u \in \langle x \rangle$.

$\langle x \rangle$ is a subset of a finite group G. But the definition of $\langle x \rangle$ involves an infinite sequence of powers of x, all of which belong to $\langle x \rangle$ since $*$ is closed. This is possible only if the powers of x start repeating. Let $m, n \in \mathbb{Z}^+$ be such that $m \neq n$ and $x^m = x^n$. There is no loss in generality in assuming that $m > n$. Let $x^{-1} \in G$ be the inverse of x. It exists because G is a group. We have not yet proved that x^{-1} belongs to $\langle x \rangle$. Multiplying both sides of the identity $x^m = x^n$ by n copies of x^{-1}, we get

$$\begin{aligned} & x^m * \underbrace{x^{-1} * \cdots * x^{-1}}_{n \text{ times}} = x^n * \underbrace{x^{-1} * \cdots * x^{-1}}_{n \text{ times}} \\ \implies & \underbrace{x * \cdots * x}_{m \text{ times}} * \underbrace{x^{-1} * \cdots * x^{-1}}_{n \text{ times}} = \underbrace{x * \cdots * x}_{n \text{ times}} * \underbrace{x^{-1} * \cdots * x^{-1}}_{n \text{ times}} \\ \implies & \underbrace{x * \cdots * x}_{m-n \text{ times}} = x^{m-n} = e. \end{aligned}$$

Since $m - n \in \mathbb{Z}^+$, x^{m-n} belongs to $\langle x \rangle$. This proves that the identity e of G also belongs to $\langle x \rangle$.

(iv) Inverse: For any $u \in \langle x \rangle$, there exists a $v \in \langle x \rangle$ such that $u*v = v*u = e$.

Let $p = m - n$ defined in the previous part. Then $x^p = e$. For $u \in \langle x \rangle$, let $u = x^k$ for some $k \in \mathbb{Z}^+$. Consider the remainder r when k is divided by p, i.e. $r = k \bmod p$. Then $0 \leq r \leq p - 1$.

If $r = 0$, then $k = pq$ for some $q \in \mathbb{Z}_+$. This implies that

$$u = x^k = x^{pq} = \underbrace{x^p * \cdots * x^p}_{q \text{ times}} = \underbrace{e * \cdots * e}_{p \text{ times}} = e.$$

In this case, the inverse of u is u itself.

If $r > 0$, then $k = pq + r$ for some non-negative integer q. Let $v = x^{p-r}$. Since $r \leq p - 1$, $p - r \in \mathbb{Z}^+$ and $v \in \langle x \rangle$. Then

$$u * v = x^k * x^{p-r} = x^{pq+r} * x^{p-r} = x^{(q+1)p} = \underbrace{x^p * \cdots * x^p}_{q+1 \text{ times}} = e.$$

Similarly, we can show that $v * u = e$. So v is the inverse of u.

□

So why are we interested in subgroups generated by elements of a finite group? If you recall our discussion of the ECDSA public key structure, the public key kP belongs to the subgroup of $E \cup \{\mathcal{O}\}$ generated by the base point P. While we have not even defined the binary operation on $E \cup \{\mathcal{O}\}$, we have enough machinery to argue that the subgroup generated by P is in fact equal to the entire group $E \cup \{\mathcal{O}\}$. We will need the following theorem.

Theorem 3. *Let G be a group whose order is a prime number. Let $x \in G$ and $x \neq e$. Then the subgroup generated by x is equal to G.*

Proof. By Lagrange's theorem, the order of the subgroup $\langle x \rangle$ generated by x divides the order of G. Let p be the order of G. Since p is a prime, the order of $\langle x \rangle$ can be either 1 or p. Since $x \neq e$, $\langle x \rangle$ has at least two elements. This implies that the order of $\langle x \rangle$ is equal to p, which in turn implies that $\langle x \rangle = G$. □

The order of the group $E \cup \{\mathcal{O}\}$ is a 256-bit prime number n. If the base point P is not the identity of $E \cup \{\mathcal{O}\}$, then by the above theorem the subgroup generated by P is equal to $E \cup \{\mathcal{O}\}$. To prove the P is not the identity of $E \cup \{\mathcal{O}\}$ will require us to define the binary operation on $E \cup \{\mathcal{O}\}$. For now, we can give an argument based on common sense. If the base point P were the identity element of $E \cup \{\mathcal{O}\}$, then all the public keys kP would be equal to P. The digital signature scheme would break down since the same public key would validate signatures created by any private key. Also, since the public keys are used to derive Bitcoin addresses, there would be only one Bitcoin address in the whole system!

So what if the subgroup generated by the base point P is equal to $E \cup \{\mathcal{O}\}$? The fact that $\langle P \rangle$ is equal to $E \cup \{\mathcal{O}\}$ ensures that distinct private keys give rise to distinct public keys. Suppose this was false and $k_1 P = k_2 P$ for $k_1, k_2 \in \{1, 2, \ldots, n-1\}$ such that $k_1 \neq k_2$. Let $-P$ be the inverse of the base point P. We will later see that the point at infinity $\mathcal{O}$ is the identity of the group $E \cup \{\mathcal{O}\}$.

Assume that $k_1 > k_2$. Adding $-P$ to both sides of the equation k_2 times gives us

$$\begin{aligned} & k_1 P + \underbrace{-P + \cdots + -P}_{k_2 \text{ times}} = k_2 P + \underbrace{-P + \cdots + -P}_{k_2 \text{ times}} \\ \implies & \underbrace{P + \cdots + P}_{k_1 \text{ times}} + \underbrace{-P + \cdots + -P}_{k_2 \text{ times}} = \underbrace{P + \cdots + P}_{k_2 \text{ times}} + \underbrace{-P + \cdots + -P}_{k_2 \text{ times}} \\ \implies & \underbrace{P + \cdots + P}_{k_1 - k_2 \text{ times}} = \mathcal{O} \implies (k_1 - k_2)P = \mathcal{O}. \end{aligned}$$

But this is a contradiction because it implies that $\langle P \rangle$ has order at most $k_1 - k_2 \leq n - 2$.

2.2 Fields

The next algebraic object we need in order to understand ECDSA is called a field. A field can be described succintly in terms of abelian groups.

Definition 4 (Abelian Group)**.** *A group G with binary operation $*$ is called an abelian group if $x * y = y * x$ for all $x, y \in G$.*

In abelian groups, the result of the binary operation between a pair of elements is independent of the order of the elements. All the examples of groups we encountered in the previous section were abelian groups. An example of a non-abelian group is the set of $n \times n$ nonsingular matrices with real entries where $n \geq 2$ and matrix multiplication is the binary operation. For instance, let $n = 2$ and consider the following calculation where A and B are 2×2 nonsingular matrices.

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}}_{B} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix} \neq \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}}_{A}$$

Definition 5 (Field)**.** *Let F be a set with two binary operations $+$ and $*$ defined on it. F is called a field if it satisfies the following properties.*

(i) F is an abelian group with respect to $+$. Let 0 denote the identity of F.

(ii) Let $F^* = F \setminus \{0\}$. F^* is an abelian group with respect to $*$.

(iii) For any $x, y, z \in F$,

$$x * (y + z) = x * y + x * z.$$

While $+$ and $*$ can be arbitrary binary operations which satisfy the required properties, they are usually referred to as "addition" and "multiplication" respectively. The third requirement in the field definition is called the distributivity of multiplication over addition.

Here are two familiar examples of fields.

- The real numbers $\mathbb{R}$ form a field with $+$ and $*$ defined as the usual addition and multiplication of real numbers.
- The rational numbers $\mathbb{Q}$ form a field with $+$ and $*$ defined as the usual addition and multiplication of rational numbers.

Both $\mathbb{R}$ and $\mathbb{Q}$ are fields with an infinite number of elements. Fields with finite cardinality are called *finite fields*. To describe the ECDSA with the `secp256k1` domain parameters, we need to define a finite field whose cardinality is a prime number. Such fields are called prime fields.

Prime Fields

Let p be a prime number. Let $\mathbb{F}_p = \{0, 1, 2, \ldots, p-1\}$ be the integers from 0 to $p-1$. Define $+$ and $*$ on $\mathbb{F}_p$ as integer addition modulo p and integer multiplication modulo p respectively, i.e. for all $x, y \in \mathbb{F}_p$

$$x + y = x + y \bmod p,$$
$$x * y = xy \bmod p.$$

We prove that $\mathbb{F}_p$ is a field.

Proof. We need to check the three properties stated in the field definition.

- Since $\mathbb{F}_p$ is the same as $\mathbb{Z}_p$ with $+$ as the binary operation, it is a group with 0 as the identity element. It is an abelian group as $x + y \bmod p = y + x \bmod p$.
- $\mathbb{F}_p^* = \mathbb{F}_p \setminus \{0\} = \{1, 2, \ldots, p-1\}$. To prove that $\mathbb{F}_p^*$ is a group with $*$ as the binary operation, we need to check closure of $*$, associativity of $*$, existence of identity and inverses. Once we have shown that $\mathbb{F}_p^*$ is a group, it will follow that it is an abelian group as $xy \bmod p = yx \bmod p$.
  - *Closure*: The result of $x * y$ is an integer from $\mathbb{F}_p$. The only way $*$ can fail to be closed on $\mathbb{F}_p^*$ is when $x * y = 0$ when $x, y \in \mathbb{F}_p^*$. But $x * y = 0$ implies that the integer product xy is a multiple of p, i.e. $xy = kp$ for some integer k. As elements of $\mathbb{F}_p^*$, x and y only have factors less than p. The product of such factors cannot be equal to p since it is a prime number.

- *Associativity*: Integer multiple modulo p is associative which makes $*$ associative.
- *Identity*: As $p \geq 2$, the integer 1 belongs to $\mathbb{F}_p^*$. It is the identity of $\mathbb{F}_p^*$ as $x * 1 = 1 * x = x$ for all $x \in \mathbb{F}_p^*$.
- *Inverses*: To prove that existence of the inverse of $x \in \mathbb{F}_p^*$, we have to find a $y \in \mathbb{F}_p^*$ such that $x * y = y * x = 1$. We will need a result from number theory which we state without proof.

  Bézout's identity: *Let x and y be integers which are not both zero. Let $\gcd(x, y)$ be the greatest common divisor of x and y. Then there exist integers u and v such that*

  $$\gcd(x, y) = xu + yv.$$

  The identity states that the gcd of two integers x, y can be written as an integer linear combination of x and y. For example, the integers 15 and 35 have gcd 5 which can be written as $5 = 35(1) + 15(-2)$. The integers u and v can be calculated from the Euclidean algorithm for finding gcds.

  To find the multiplicative inverse of $x \in \mathbb{F}_p^*$, consider the gcd of x and p. Since x is a positive integer less than p and p is a prime, $\gcd(x, p) = 1$. By Bézout's identity, there exist integers u, v such that $xu + pv = 1$. Considering both sides of this equation modulo p, we get $xu \bmod p = 1$.

  If the integer u belongs to $\mathbb{F}_p^*$, then it is the inverse of x as $x * u = u * x = 1$. If $u \notin \mathbb{F}_p^*$, then divide it by p to get the remainder r, i.e. $u = qp + r$ where $0 \leq r \leq p - 1$. Note that r cannot be 0 as this would mean u is a multiple of p and $xu \bmod p = 0$. So r belongs to $\mathbb{F}_p^*$ and is the inverse of x as

  $$x * r = r * x = xr \bmod p = x(u - qp) \bmod p = xu \bmod p = 1.$$

- Distributivity of $*$ over $+$ follows as

  $$\begin{aligned} x * (y + z) &= x(y + z \bmod p) \bmod p = x(y + z) \bmod p \\ &= xy \bmod p + xz \bmod p = x * y + x * z. \end{aligned}$$

□

To illustrate the structure of a prime field, consider $p = 5$. Table 2.1 shows the results of addition and multiplication of elements in $\mathbb{F}_5$.

The theory of finite fields states that any finite field has cardinality equal to a prime power p^m. When $m = 1$, we get prime fields. We do not describe the structure of finite fields with $m > 1$ since the ECDSA used in Bitcoin only involves a prime field.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Table 2.1: Addition and multiplication operations in $\mathbb{F}_5$

2.3 Elliptic Curves over Real Numbers

The ECDSA is defined using elliptic curves over finite fields. However, elliptic curves over the real numbers are easier to visualize. In this section, we discuss elliptic curves over the reals which will help us understand these curves over finite fields.

Let $a, b \in \mathbb{R}$ such that $4a^3 + 27b^2 \neq 0$. Let E be the set of real solutions (x, y) of the equation

$$y^2 = x^3 + ax + b. \tag{2.6}$$

Figure 2.2 illustrates two examples of the set E. An elliptic curve over the real numbers is the set $E \cup \{\mathcal{O}\}$ where $\mathcal{O}$ is called the point at infinity. A precise definition of $\mathcal{O}$ requires concepts from projective geometry which we will not discuss. The utility of $\mathcal{O}$ will become clear when we define a binary operation on $E \cup \{\mathcal{O}\}$ and endow it with a group structure.

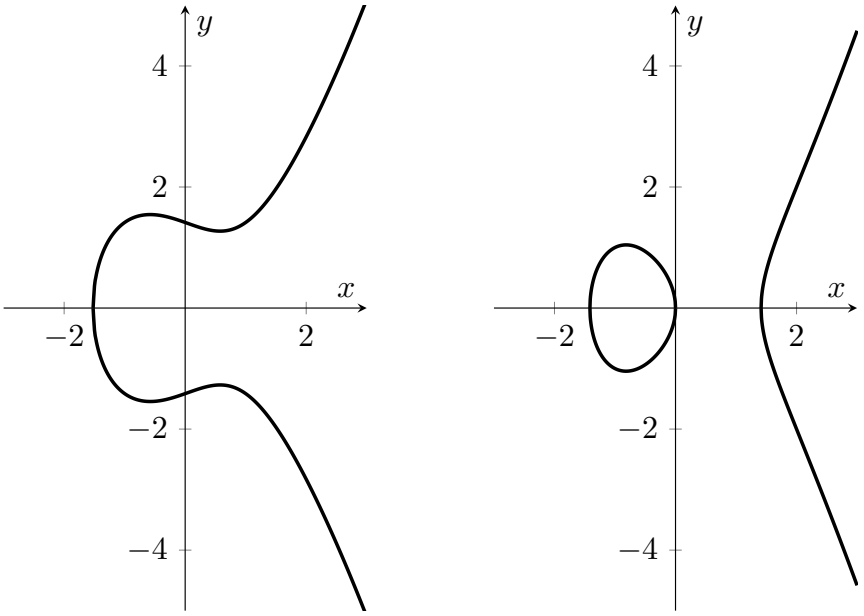
The condition $4a^3 + 27b^2 \neq 0$ ensures that the cubic $x^3 + ax + b$ does not have multiple roots. It is analogous to the condition $b^2 - 4ac \neq 0$ which ensures that the quadratic $ax^2 + bx + c$ does not have multiple roots. Curves defined by equation (2.6) for which $4a^3 + 27b^2 = 0$ are called singular curves and are excluded from the class of elliptic curves by definition. One reason is that singular curves contain points in E which prevent $E \cup \{\mathcal{O}\}$ from being a group.

Group Structure of $E \cup \{\mathcal{O}\}$

The first step in showing $E \cup \{\mathcal{O}\}$ is a group is to define a binary operation on it. We will refer to the operation as "point addition" and denote it by $+$.

Let us disregard $\mathcal{O}$ for the moment and consider points $P, Q \in E$. Then $P = (x_1, y_1) \in \mathbb{R}^2$ and $Q = (x_2, y_2) \in \mathbb{R}^2$ such that (x_1, y_1) and (x_2, y_2) are solutions to equation (2.6). We want to find the coordinates (x_3, y_3) of the point $R = P + Q$.

Suppose the line through P and Q intersects the curve again at a point R' as illustrated in Figure 2.3(a). Let R be the reflection of R' about the x-axis. Set $P + Q = Q + P = R$. But what if the line through P and Q never intersects the curve again as illustrated in Figure 2.3(b)? And what about the



(a) $y^2 = x^3 - x + 2$ (b) $y^2 = x^3 - 2x$

Figure 2.2: Examples of elliptic curves over $\mathbb{R}$

case when $P = Q$? Which line through P should we consider? We will define point addition differently in these cases. For now, let us assume that P and Q are distinct points not on the same vertical line, i.e. $x_1 \neq x_2$. For $x_1 \neq x_2$, the line through the points (x_1, y_1) and (x_2, y_2) is given by

$$y = mx + y_1 - mx_1$$

where $m = \frac{y_2 - y_1}{x_2 - x_1}$. To find the points of intersection between this line and the curve given in equation (2.6), let us eliminate y from the curve equation by substituting the expression for y from the line. We get

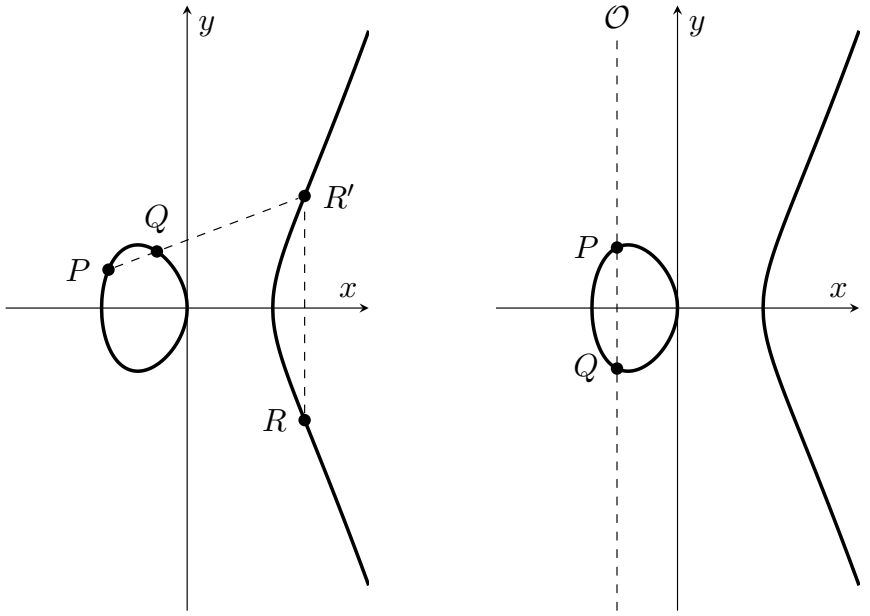
$$\begin{aligned} &(mx + y_1 - mx_1)^2 = x^3 + ax + b \\ \implies &x^3 - m^2x^2 + [a - 2m(y_1 - mx_1)]\, x + b - (y_1 - mx_1)^2 = 0. \end{aligned} \tag{2.7}$$

The solutions to the above cubic equation will be the x coordinates of the points of intersection between the line and the elliptic curve. We already know two such coordinates, namely x_1 and x_2. If we denote the x coordinate of the third intersection point R' by x'_3, another representation for the cubic in equation (2.7) is

$$(x - x_1)(x - x_2)(x - x'_3) = 0.$$

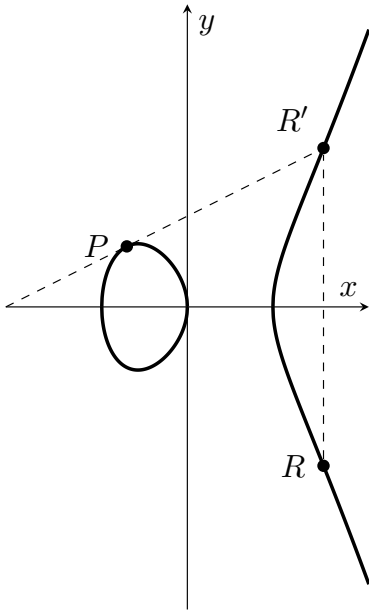
Equating the coefficients of x^2 in the two representations of the cubic, we get

$$m^2 = x_1 + x_2 + x'_3 \implies x'_3 = m^2 - x_1 - x_2.$$



(a) $x_1 \neq x_2$

(b) $x_1 = x_2, y_1 = -y_2$



(c) $x_1 = x_2, y_1 = y_2 \neq 0$

Figure 2.3: Point addition in elliptic curves over $\mathbb{R}$

Substituting x_3' into the line equation, we get the y coordinate of the third intersection point R' as

$$y_3' = mx_3' + y_1 - mx_1.$$

Since R is the reflection of R' about the x-axis, its coordinates are $(x_3, y_3) = (x'_3, -y'_3)$.

Now consider the case when P and Q are distinct points on the curve which lie on a vertical line as shown in Figure 2.3(b). The coordinates of P and Q are related as $x_1 = x_2$ and $y_1 = -y_2$. In this case, we define $P+Q$ to be equal to the point at infinity $\mathcal{O}$. It is convenient to think of $\mathcal{O}$ as a point which lies at infinite height along the line joining P and Q. With this interpretation of $\mathcal{O}$, the line joining any $P \in E$ and $\mathcal{O}$ will be a vertical line which intersects the curve again at Q, the reflection of P about the x-axis. The reflection of Q will be P itself. So by the intersection-reflection procedure for point addition we described earlier, we get $P + \mathcal{O} = \mathcal{O} + P = P$. This almost makes $\mathcal{O}$ the identity element of $E \cup \{\mathcal{O}\}$ under point addition. We say almost because the case of $P = \mathcal{O}$ is not handled as P was assumed to be a point in E. To make $\mathcal{O}$ into the identity element, we will define $\mathcal{O} + \mathcal{O} = \mathcal{O}$.

If $\mathcal{O}$ is the identity element of $E \cup \{\mathcal{O}\}$, then the inverse of a point $P \in E$ is its reflection Q about the x-axis since we defined $P + Q = Q + P = \mathcal{O}$. We will denote the inverse of P by $-P$. For $P = (x, y)$, we have $-P = (x, -y)$. The inverse of $\mathcal{O}$ is $\mathcal{O}$ itself.

Finally, let us consider the case when P and Q are not distinct points, i.e. $P = Q$. The point addition in this case is called *point doubling* as $P + P$ is denoted as $2P$. We will apply the intersection-reflection procedure using the tangent line at P as illustrated in 2.3(c). The resulting point R will be defined to be equal to $2P$. For $P = (x_1, y_1)$, the slope of the tangent to the curve at P is given by

$$m_1 = \frac{dy}{dx} = -\left.\frac{\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial y}}\right|_{x=x_1, y=y_1} = \frac{3x_1^2 + a}{2y_1}$$

where $f(x, y) = y^2 - x^3 - ax - b$. If $y_1 = 0$, the tangent is a vertical line which does not intersect the curve again. In this case, we define $2P = \mathcal{O}$. If $y_1 \neq 0$, to find the second point of intersection of the tangent and the curve, we eliminate y from equation (2.6) using the tangent equation

$$y = m_1 x + y_1 - m_1 x_1.$$

The resulting cubic equation is given by

$$g(x) = (m_1 x + y_1 - m_1 x_1)^2 - x^3 - ax - b = 0.$$

The polynomial $g(x)$ has x_1 as a root since $P = (x_1, y_1)$ lies on both the tangent and the curve. It turns out that x_1 is a double root as the derivative of $g(x)$ evaluated at x_1 is zero.

$$\begin{aligned}\left.\frac{dg}{dx}\right|_{x=x_1} &= 2m_1 (m_1 x_1 + y_1 - m_1 x_1) - 3x_1^2 - a \\ &= 2m_1 y_1 - 3x_1^2 - a = 0.\end{aligned}$$

If we denote the x coordinate of the second intersection point R' by x_2', another representation for $g(x)$ is

$$(x - x_1)^2(x - x_2') = 0.$$

Equating the coefficients of x^2 in the two representations of $g(x)$, we get

$$m_1^2 = 2x_1 + x_2' \implies x_2' = m_1^2 - 2x_1.$$

Substituting x_2' into the line equation, we get the y coordinate of R' as

$$y_2' = m_1 x_2' + y_1 - m_1 x_1.$$

The reflection R of R' about the x-axis has coordinates $(x_2, y_2) = (x_2', -y_2')$.

We now summarize the point addition operation on $E \cup \{\mathcal{O}\}$.

Definition 6 (Point Addition)**.** *Let E be the set of real solutions (x, y) of the equation $y^2 = x^3 + ax + b$ where $4a^3 + 27b^2 \neq 0$. Let $\mathcal{O} \notin E$ be a special element called the point at infinity. Then the binary operation $+$ on $E \cup \{\mathcal{O}\}$ is defined as follows.*

(i) If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $x_1 \neq x_2$, then $P + Q = (x_3, y_3)$ where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2, \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \quad (2.8)$$

(ii) If $P = (x_1, y_1)$ and $Q = (x_1, -y_1)$, then $P + Q = \mathcal{O}$.

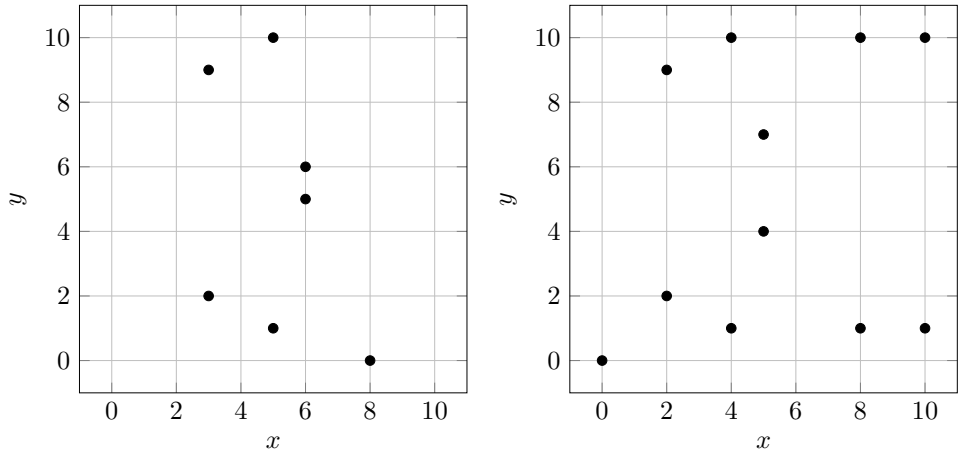
(iii) If $P = (x_1, y_1) \in E$ with $y_1 \neq 0$, then $P + P = (x_2, y_2)$ where

$$x_2 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1, \quad y_2 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_2) - y_1. \quad (2.9)$$

(iv) For any $P \in E \cup \{\mathcal{O}\}$, $P + \mathcal{O} = \mathcal{O} + P = P$.

Note that the case of point doubling when $y_1 = 0$ is subsumed by the second case above as $(x_1, y_1) = (x_1, -y_1)$ for $y_1 = 0$.

So have we proved that $E \cup \{\mathcal{O}\}$ is a group under the point addition operation? The operation is closed by construction, $\mathcal{O}$ is the identity element and every element in $E \cup \{\mathcal{O}\}$ has an inverse. The only group property remaining to be checked is the associativity of point addition. While $+$ is indeed associative, we will skip the proof as it is a tedious exercise to prove associativity using the above rules.



(a) $y^2 = x^3 + 10x + 2$ (b) $y^2 = x^3 + 9x$

Figure 2.4: Examples of elliptic curves over $\mathbb{F}_{11}$

2.4 Elliptic Curves over Finite Fields

Let F be a finite field whose cardinality is not divisible by 2 or 3. In this case, the definition of an elliptic curve over F is similar to its definition over real numbers. Let $a, b \in F$ such that $4a^3 + 27b^2 \neq 0$. Let E be the set of points (x, y) in $F \times F$ which satisfy the equation $y^2 = x^3 + ax + b$. An elliptic curve over F is the set $E \cup \{\mathcal{O}\}$ where $\mathcal{O}$ is the point at infinity.

When the cardinality of F is divisible by either 2 or 3, the curve equation of an elliptic curve over F is different and so is the condition on the coefficients. We will not discuss this case as the ECDSA used in Bitcoin uses an elliptic curve defined over the prime field $\mathbb{F}_p$ where p is the 256-bit prime given in equation (2.1). The cardinality p of $\mathbb{F}_p$ is obviously not divisible by 2 or 3.

Figure 2.4 shows the solutions to the equations $y^2 = x^3 + 10x + 2$ and $y^2 = x^3 + 9x$ over $\mathbb{F}_{11} = \{0, 1, 2, \ldots, 10\}$. Since the additive inverses of 1 and 2 in $\mathbb{F}_{11}$ are 10 and 9 respectively, these curves are the same as those considered in Figure 2.2 with the coefficients interpreted as elements in $\mathbb{F}_{11}$. While these curves do not resemble their counterparts over the real numbers, there is horizontal symmetry in the solution sets of both curves. For every solution (x, y), there is a solution at $(x, 11 - y)$. This is because $(11 - y)^2 \bmod 11 = y^2 \bmod 11$. The pair of solutions (x, y) and $(x, 11 - y)$ coincide for $y = 0$ as $11 = 0 \bmod 11$. Consequently, the points $(8, 0)$ in Figure 2.4(a) and $(0, 0)$ in Figure 2.4(b) violate the horizontal symmetry.

The point addition operation for elliptic curves over F can be defined as in Definition 6 with the expressions in equations (2.8) and (2.9) interpreted as operations on elements from F. Subtraction of a field element x is interpreted as addition of its additive inverse $-x$ and division by a non-zero field element x is interpreted as multiplication by its multiplicative inverse x^{-1}.

$+$	$\mathcal{O}$	$(3,2)$	$(3,9)$	$(5,1)$	$(5,10)$	$(6,5)$	$(6,6)$	$(8,0)$
$\mathcal{O}$	$\mathcal{O}$	$(3,2)$	$(3,9)$	$(5,1)$	$(5,10)$	$(6,5)$	$(6,6)$	$(8,0)$
$(3,2)$	$(3,2)$	$(6,6)$	$\mathcal{O}$	$(6,5)$	$(8,0)$	$(3,9)$	$(5,10)$	$(5,1)$
$(3,9)$	$(3,9)$	$\mathcal{O}$	$(6,5)$	$(8,0)$	$(6,6)$	$(5,1)$	$(3,2)$	$(5,10)$
$(5,1)$	$(5,1)$	$(6,5)$	$(8,0)$	$(6,6)$	$\mathcal{O}$	$(5,10)$	$(3,9)$	$(3,2)$
$(5,10)$	$(5,10)$	$(8,0)$	$(6,6)$	$\mathcal{O}$	$(6,5)$	$(3,2)$	$(5,1)$	$(3,9)$
$(6,5)$	$(6,5)$	$(3,9)$	$(5,1)$	$(5,10)$	$(3,2)$	$(8,0)$	$\mathcal{O}$	$(6,6)$
$(6,6)$	$(6,6)$	$(5,10)$	$(3,2)$	$(3,9)$	$(5,1)$	$\mathcal{O}$	$(8,0)$	$(6,5)$
$(8,0)$	$(8,0)$	$(5,1)$	$(5,10)$	$(3,2)$	$(3,9)$	$(6,6)$	$(6,5)$	$\mathcal{O}$

Table 2.2: Point addition for the elliptic curve $y^2 = x^3 + 10x + 2$ over $\mathbb{F}_{11}$

For $x_1, y_1, x_2, y_2 \in F$ with $x_1 \neq x_2$, equation (2.8) can be written in terms of the field operations $+$ and $*$ as

$$\begin{aligned} x_3 &= [y_2 + (-y_1)]^2 * [x_2 + (-x_1)]^{-2} + (-x_1) + (-x_2) \\ y_3 &= [y_2 + (-y_1)] * [x_2 + (-x_1)]^{-1} * [x_1 + (-x_3)] + (-y_1). \end{aligned} \tag{2.10}$$

where $x^2 = x * x$ and $x^{-2} = x^{-1} * x^{-1}$. For $x_1, y_1 \in F$ with $y_1 \neq 0$, equation (2.9) can be written as

$$\begin{aligned} x_2 &= \left(3 * x_1^2 + a\right)^2 * (2 * y_1)^{-2} + (-2 * x_1) \\ y_2 &= \left(3 * x_1^2 + a\right) * (2 * y_1)^{-1} * [x_1 + (-x_2)] + (-y_1). \end{aligned} \tag{2.11}$$

The point addition operation for the curve in Figure 2.4(a) is illustrated in Table 2.2. For example, to find the result of adding $(3,2)$ to $(3,9)$, we find the entry in the table which lies both in the row starting with $(3,2)$ and the column starting with $(3,9)$. The entry in this case is $\mathcal{O}$ as $-2 = 9$ in $\mathbb{F}_{11}$.

The `secp256k1` Elliptic Curve

At the expense of some repetition, let us revisit the elliptic curve described by the `secp256k1` domain parameters. Let p be the 256-bit prime number given in equation (2.1). Let $\mathbb{F}_p$ be the corresponding prime field with operations $+$ and $*$. The curve $y^2 = x^3 + 7$ corresponds to coefficients $a = 0$ and $b = 7$ in the general equation $y^2 = x^3 + ax + b$. For these coefficients, $4a^3 + 27b^2 = 27 * 49 = 1323$ which is not equal to 0 modulo p. The curve has $n-1$ solutions $(x, y) \in \mathbb{F}_p^2$ where n is the 256-bit prime given in equation (2.3). Adjoining the point at infinity $\mathcal{O}$ to this set E of solutions, we get the set $E \cup \{\mathcal{O}\}$ of cardinality n. $E \cup \{\mathcal{O}\}$ is a group with respect to the point addition operation defined in Definition 6 where equations (2.8) and (2.9) are interpreted as equations (2.10) and (2.11) respectively. The additive inverse $-x$ of $x \in \mathbb{F}_p$ is 0 if $x = 0$ and $p - x$ if $x \neq 0$. The multiplicative inverse x^{-1} of $x \in \mathbb{F}_p^*$ is obtained using the Euclidean algorithm.

A private key k is an integer in the range $\{1, 2, \ldots, n-1\}$. It can be specified using 256 bits (32 bytes) as n is a 256-bit number. The corresponding public key kP is obtained by adding k copies of the base point $P \in E$ given in equation (2.4). As P is a point in $\mathbb{F}_p^2$, it is not equal to the identity $\mathcal{O}$. Since the order of the group $E \cup \{\mathcal{O}\}$ is prime, the subgroup $\langle P \rangle$ generated by P is equal to $E \cup \{\mathcal{O}\}$. This implies that $kP \neq \mathcal{O}$ for $1 \leq k \leq n-1$ and $nP = \mathcal{O}$. So a public key is always a point (x, y) in $\mathbb{F}_p^2$ and can be specified using 512 bits (64 bytes) as p is a 256-bit number.

If we fix $x = x_1 \in \mathbb{F}_p$, the equation $y^2 = x_1^2 + 7$ is a quadratic equation in y and can have at most two solutions. If $y_1 \in \mathbb{F}_p$ is one solution, the other solution is given by $p - y_1$ as $y_1^2 = (p - y_1)^2 \bmod p$. As p is an odd integer, one of these solutions is an odd integer and the other is an even integer. This fact is exploited to reduce the size of a public key to 33 bytes. In the *compressed public key* format, the public key (x, y) is represented by a single byte followed by the 32-byte x coordinate. The single byte at the beginning is set to `0x02` or `0x03` depending on whether the y coordinate is even or odd. In the *uncompressed public key* format, the public key consists of a single byte containing `0x04` followed by the 32 byte x coordinate and the 32 byte y coordinate, resulting in a key size of 65 bytes.

2.5 ECDSA

Signing a message using a digital signature algorithm involves the creation of a bit string which is easy to create if a private key is known and computationally infeasible otherwise. The created bit string is called a digital signature. It is attached to the message as proof that an entity with knowledge of a private key has signed the message (see Figure 2.1). Unlike the usual handwritten signatures, digital signatures are message dependent. If a digital signature did not depend on the message being signed, it could be attached to a different message which the signer did not sign and still serve as a valid signature. But plain dependence on the messages being signed is not enough. A digital signature algorithm needs to be resistant to forgery. Informally, unforgeability requires that an adversary with access to signatures on a set of messages created using a private key should not be able to create a valid signature on a new message in a computationally feasible manner.

The unforgeability of the digital signatures created using the ECDSA is due to the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP). The ECDLP refers to the problem of finding a private key k given the public key kP for elliptic curves defined over finite fields. The original discrete logarithm problem (DLP) was defined using a prime field $\mathbb{F}_p$. It involves finding the positive integer x given $a, b \in \mathbb{F}_p$ such that $a^x = b \bmod p$. Here x can be intepreted as the discrete logarithm of b to the base a. Since a^x is the result of combining x copies of a with the multiplication operation,

it is analogous to kP which is the result of combining k copies of P using the point addition operation. Finding k given P and kP is thus interpreted as the elliptic curve analogue of the DLP.

For brevity, we will avoid a general description of ECDSA and describe it as it is used in Bitcoin. Let m be a bit string representing the message to be signed. The length of this message can be at most $2^{64} - 1$ bits. Let e be the result of applying the SHA-256 hash function on m twice, i.e. $e =$ SHA-256(SHA-256(m)). The SHA-256 hash function and its properties will be considered in Chapter 3. For now, consider it as a function which takes as input any bit string of length at most $2^{64}-1$ bits and returns a 256-bit output. Thus the bit string e, which is called the *message digest*, will be 256 bits long. Let P be the base point specified by the `secp256k1` domain parameters. For a private key k, the corresponding digital signature on m is generated as follows:

1. Choose a random integer j from $\{1, 2, \ldots, n-1\}$ where n is the prime from equation (2.3).

2. Add the base point P to itself j times to get $jP = (x, y)$.

3. Calculate $r = x \bmod n$. If $r = 0$, go to step 1.

4. Calculate the multiplicative inverse j^{-1} of j in the prime field $\mathbb{F}_n$.

5. Calculate $s = j^{-1}(e + kr) \bmod n$ where the message digest e is interpreted as an integer. If $s = 0$, go to step 1.

6. Output (r, s) as the signature for the message m.

Some discussion is in order before we present the signature verification procedure.

- As n is a 256-bit integer, the signature (r, s) is 512 bits long.

- Due to the random choice of j, two different runs of the signature generation algorithm for the same message m will yield different signatures.

- The motivation behind using a message digest e instead of the message m is that we want the signature to depend on the entire message and still have a small length. If we were to replace e with m in the calculation of s, i.e. $s = j^{-1}(m+kr) \bmod n$, then s would depend only on $m \bmod n$. Any message m' which differs from m in a multiple of n will have the same signature, which violates the unforgeability requirement. An alternative to signing the message digest is to consider the base n representation of the message m, i.e. $m = a_0 + a_1 n + a_2 n^2 + \cdots$ where $0 \leq a_i \leq n-1$. We could then generate a signature for each digit a_i but this would result in long signatures.

- Since e is an arbitrary 256-bit string and $n < 2^{256} - 1$, it is possible that e is strictly greater than n. In that case, $e \bmod n \neq e$. Then the signatures on messages with digests e and $e \bmod n$ will be the same. Is this not a violation of the unforgeability requirement? To be able to forge a signature using this property, an adversary would have to find a message m' whose message digest is equal to $e \bmod n$. This is computationally infeasible under the assumption that the SHA-256 hash function is resistant to preimage attacks (see Chapter 3). So the unforgeability requirement is not violated.

- In step 3, if r is allowed to be zero then in step 5 the signature would become independent of the private key k.

- In step 4, the multiplicative inverse j^{-1} exists as $j \neq 0$.

- In step 5, s is not allowed to be zero because its multiplicative inverse s^{-1} is essential for signature verification.

- Suppose an adversary who has the access to the message m and the corresponding signature (r, s) wants to recover the private key k. Since m is known, the adversary can calculate the message digest e. If the adversary can find j, then the private key k can be calculated as $k = r^{-1}(js - e) \bmod n$. Finding j from r involves finding j given jP, i.e. solving the ECDLP which is computationally infeasible.

Given a message m, a public key kP, and a digital signature (r, s), the signature is verified as follows:

1. Calculate the message digest $e = \text{SHA-256}(\text{SHA-256}(m))$.

2. Calculate the multiplicative inverse s^{-1} of s in the prime field $\mathbb{F}_n$.

3. Calculate $j_1 = es^{-1} \bmod n$ and $j_2 = rs^{-1} \bmod n$.

4. Calculate the point $Q = j_1P + j_2(kP)$. Here $j_2(kP)$ represents the result of adding j_2 copies of the public key kP.

5. If $Q = \mathcal{O}$, then the signature is invalid.

6. If $Q \neq \mathcal{O}$, then let $Q = (x, y) \in \mathbb{F}_p^2$. Calculate $t = x \bmod n$. If $t = r$, the signature is valid.

To see why the above verification procedure works, consider a valid signature (r, s) on a message m with digest e. Then there exists an integer $j \in \{1, 2, \ldots, n-1\}$ such that $jP = (x_1, y_1)$, $r = x_1 \bmod n$, and $s = j^{-1}(e + kr)$. The integer j satisfies $j = s^{-1}(e + kr)$ which give us

$$Q = j_1P + j_2(kP) = (j_1 + j_2k)P = (es^{-1} + rs^{-1}k)P = s^{-1}(e + kr)P = jP.$$

In the above equation, we could suppress the $\mod n$ expressions while substituting for j_1 and j_2 because $nP = \mathcal{O}$ and $\mathcal{O}$ is the identity for point addition. Since $Q = jP$, their x coordinates are equal modulo n. Since $j \in \{1, 2, \ldots, n-1\}$, jP is never equal to $\mathcal{O}$. This is the reason behind rejecting the signature in step 5.

Chapter 3

Cryptographic Hash Functions

Hash functions are defined as functions which map bit strings of arbitrary length to bit strings of fixed length. As the number of possible inputs is larger than the number of possible outputs, a hash function is a many-to-one function. A cryptographic hash function H is defined as a hash function which has the following properties:

- **Preimage resistance:** Let y be the output of H for some unknown input. An input x which satisfies $H(x) = y$ is called a preimage of y under H. By the many-to-one nature of H, preimages are not necessarily unique. The function H is said to be preimage resistant if it is computationally infeasible to calculate any preimage x of a given y.
- **Second preimage resistance:** Given an input string x, it is computationally infeasible to find a different input string x' such that $H(x) = H(x')$.
- **Collision resistance:** It is computationally infeasible to find a pair of distinct input strings x, x' such that $H(x) = H(x')$.

Preimage resistance captures the requirement that a cryptographic hash function needs to be difficult to invert. While the second preimage resistance and collision resistance properties look similar, the difference lies in the choice of the input x. In the second preimage resistance property, the input x is fixed and an adversary is required to find another input x' such that $H(x') = H(x)$. On the other hand, in the collision resistance property the adversary has the freedom to choose both x and x'.

The parity function, which maps bit strings having an even number of ones to 0 and bit strings having an odd number of ones to 1, is technically a hash function. But it is not a cryptographic hash function as all three required properties are easily violated. Any other hash function whose output length

n is small will also fail to be a cryptographic hash function as collisions can be found by calculating the outputs corresponding to $2^n + 1$ different inputs. This follows from the pigeonhole principle. *How large does n need to be?* Typically n is of the order of hundreds of bits. For instance, Bitcoin uses the SHA-256 and RIPEMD-160 hash functions which have output lengths of 256 and 160 bits respectively. But having a large output length is not sufficient for a hash function to be considered cryptographic. The relation between the input bit string x and the output bit string y should be complicated to prevent easy recovery of x from y. *When can an input-output relationship be considered complicated?* While this is a difficult question to answer in general, we present the design of SHA-256 in this chapter as an example of a complicated input-output relationship.

The SHA-256 hash function plays a crucial role in the design of the Bitcoin system. In addition to its use in creating message digests in the ECDSA, it is used to control the creation of new bitcoins by the nodes in the Bitcoin P2P network. Both SHA-256 and RIPEMD-160 are used to create Bitcoin addresses from ECDSA public keys. While there is no formal proof that these two hash functions are in fact cryptographic hash functions, they have no known weaknesses which make finding preimages, second preimages, and collisions computationally feasible.

3.1 SHA-256

The SHA-256 hash function was announced in 2001 by the National Institute of Standards and Technology (NIST), an agency which is part of the U.S. Department of Commerce. It was specified as part of the Secure Hash Standard[1] detailed in the Federal Information Processing Standards Publication 180-2 (FIPS PUB 180-2). The SHA in SHA-256 is an abbreviation of "secure hash algorithm" and the 256 indicates the output length in bits.

While the definition of a hash function allowed its input to be a bit string of arbitrary length, the SHA-256 function specification restricts the input to be at most $2^{64} - 1$ bits long. This restriction is imposed because the specification requires the length of the input to be stored in a 64-bit unsigned integer. The SHA-256 operation can be divided into preprocessing and hash computation. For convenience, let us refer to the input bit string as the message and denote it by M.

Preprocessing

The preprocessing step consists of message padding and state initialization. Message padding involves appending some bits to the message resulting in a

[1] See `http://dx.doi.org/10.6028/NIST.FIPS.180-4` for the latest version of this standard

padded message whose length is a multiple of 512. It proceeds as follows:

1. If the message M is l bits long, then find the smallest non-negative solution k to the equation
$$k + l + 65 = 0 \bmod 512. \tag{3.1}$$

2. Append the $(k+1)$-bit string containing a single 1 followed by k zeros to M.

3. Append the 64-bit unsigned integer representation of the message length l to the output of step 2.

Since the message M is l bits long, the padded message after step 2 is $l+k+1$ bits long. After step 3, the padded message is $l + k + 65$ bits long. The requirement that the final padded message length be a multiple of 512 is satisfied by choosing k according to equation (3.1). Note that padding is done even if the original message length l was already a multiple of 512. We will discuss the reasoning behind the choice of this particular padding scheme after we discuss the hash computation step. As an example, consider the 6-bit message $M = 101010$. For $l = 6$, the smallest non-negative solution to equation (3.1) is $k = 441$. The 64-bit representation of 6 is $00 \cdots 00110$. The 512-bit padded message is given by

$$\underbrace{101010}_{M}\ 1\ \underbrace{00000 \cdots 00000}_{441\text{ zeros}}\ \underbrace{00 \cdots 00110}_{l}.$$

State initialization involves setting the value of a 256-bit initial hash value $H^{(0)}$ to a fixed constant. Let $H_0^{(0)}, H_1^{(0)}, \ldots, H_7^{(0)}$ be eight 32-bit words which constitute $H^{(0)}$. They are initialized as follows.

$$\begin{aligned} H_0^{(0)} &= \texttt{0x6a09e667}, & H_1^{(0)} &= \texttt{0xbb67ae85}, \\ H_2^{(0)} &= \texttt{0x3c6ef372}, & H_3^{(0)} &= \texttt{0xa54ff53a}, \\ H_4^{(0)} &= \texttt{0x510e527f}, & H_5^{(0)} &= \texttt{0x9b05688c}, \\ H_6^{(0)} &= \texttt{0x1f83d9ab}, & H_7^{(0)} &= \texttt{0x5be0cd19}. \end{aligned} \tag{3.2}$$

These initial values were arbitrarily chosen by the SHA-256 designers to be the first 32 bits in the fractional parts of the square roots of the first eight prime numbers $2, 3, 5, 7, 11, 13, 17, 19$. For instance, the fractional part of $\sqrt{2}$ is $0.4142135623\cdots$. The first 32 bits in the binary representation of this fractional part are equal to `0x6a09e667` in hexadecimal representation.

Hash Computation

Let the padded message consist of N 512-bit blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. The hash computation is an iterative process where the ith message block

$M^{(i)}$ is combined with the previous hash value $H^{(i-1)}$ using a function f to generate the next hash value $H^{(i)}$.

$$H^{(i)} = f(M^{(i)}, H^{(i-1)}), \quad 1 \leq i \leq N. \tag{3.3}$$

The function f is called a *compression function* as it converts a 768-bit input to a 512-bit output. The initial hash value $H^{(0)}$ is given in equation (3.2). The final hash value $H^{(N)}$ is the output of the SHA-256 function or the *message digest* for input message M. The variable $H^{(i-1)}$ is called the *chaining variable* as it preserves the state of the hash computation across successive invocations of the compression function.

In order to specify the compression function f, we need to define some operators on 32-bit words. In what follows, U, V, and W are 32-bit words.

- Let $U \wedge V, U \vee V$, and $U \oplus V$ denote the bitwise logical AND, OR, and XOR of U and V respectively.
- Let $U + V$ denote the integer sum modulo 2^{32} of U and V.
- Let $\neg U$ denote the bitwise complement of U.
- For $U = u_0 u_1 \cdots u_{30} u_{31}$ and $1 \leq n \leq 32$, the shift right and rotate right operations on U are defined as

$$\begin{aligned} \mathrm{SHR}^n(U) &= \underbrace{000\cdots000}_{n \text{ zeros}} u_0 u_1 \cdots u_{30-n} u_{31-n}, \\ \mathrm{ROTR}^n(U) &= u_{31-n+1} u_{31-n+2} \cdots u_{30} u_{31} u_0 u_1 \cdots u_{30-n} u_{31-n}, \end{aligned}$$

  respectively.

- Let

$$\begin{aligned} \mathrm{Ch}(U,V,W) &= (U \wedge V) \oplus (\neg U \wedge W), \\ \mathrm{Maj}(U,V,W) &= (U \wedge V) \oplus (U \wedge W) \oplus (V \wedge W), \end{aligned}$$

  where Ch and Maj are short for Choice and Majority. The Ch function performs a bitwise choice between the bits of V and W depending on whether the corresponding bit in V is 1 or 0. The Maj function finds the bitwise majority among the bits of U, V, and W.

- Let

$$\begin{aligned} \Sigma_0(U) &= \mathrm{ROTR}^2(U) \oplus \mathrm{ROTR}^{13}(U) \oplus \mathrm{ROTR}^{22}(U) \\ \Sigma_1(U) &= \mathrm{ROTR}^6(U) \oplus \mathrm{ROTR}^{11}(U) \oplus \mathrm{ROTR}^{25}(U) \\ \sigma_0(U) &= \mathrm{ROTR}^7(U) \oplus \mathrm{ROTR}^{18}(U) \oplus \mathrm{SHR}^3(U) \\ \sigma_1(U) &= \mathrm{ROTR}^{17}(U) \oplus \mathrm{ROTR}^{19}(U) \oplus \mathrm{SHR}^{10}(U) \end{aligned}$$

The compression function f maintains an internal state consisting of sixty four 32-bit words $\{W_j \mid j = 0, 1, \ldots, 63\}$ which are together called the *message schedule*. It also uses 64 constant 32-bit words $K_0, K_1, \ldots, K_{63}$ which are equal to the first 32 bits in the fractional parts of the cube roots of the first 64 prime numbers $2, 3, 5, \ldots, 307, 311$. The calculation $f(M^{(i)}, H^{(i-1)})$ proceeds as follows:

1. Let $M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)}$ denote the sixteen 32-bit blocks which constitute the ith 512-bit message block $M^{(i)}$. The message schedule is initialized as follows:

$$W_j = \begin{cases} M_j^{(i)} & 0 \leq j \leq 15, \\ \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16} & 16 \leq j \leq 63. \end{cases}$$

2. Let A, B, C, D, E, F, G, H be eight 32-bit words which are initialized to the eight 32-bit words $H_0^{(i-1)}, H_1^{(i-1)}, \ldots, H_7^{(i-1)}$ which constitute the $(i-1)$th hash value $H^{(i-1)}$.

$$(A, B, C, D, E, F, G, H) = \left(H_0^{(i-1)}, H_1^{(i-1)}, \ldots, H_6^{(i-1)}, H_7^{(i-1)}\right).$$

3. For $j = 0, 1, \ldots, 63$, the variables $A, B, \ldots, H$ are iteratively updated as follows:

$$\begin{aligned} T_1 &= H + \Sigma_1(E) + \mathrm{Ch}(E, F, G) + K_j + W_j \\ T_2 &= \Sigma_0(A) + \mathrm{Maj}(A, B, C) \\ (A, B, C, D, E, F, G, H) &= (T_1 + T_2, A, B, C, D + T_1, E, F, G) \end{aligned} \tag{3.4}$$

   Each run of the above equation is called a *round*. In each of the 64 rounds, the calculation of the variable T_1 involves the constant K_j and the message schedule block W_j.

4. The eight 32-bit words $H_0^{(i)}, H_1^{(i)}, \ldots, H_7^{(i)}$ which constitute the ith hash value $H^{(i)}$ are calculated as

$$(H_0^{(i)}, H_1^{(i)}, \ldots, H_7^{(i)}) = \left(A + H_0^{(i-1)}, B + H_1^{(i-1)}, \ldots, H + H_7^{(i-1)}\right).$$

For a message M which after padding consists of N 512-bit blocks $M^{(1)}$, $M^{(2)}$,..., $M^{(N)}$, the SHA-256 hash of M is the final hash value $H^{(N)} = f(M^{(N)}, H^{(N-1)})$.

Properties

While the SHA-256 hash computation looks complicated, it can be efficiently implemented as it consists of simple operations of 32-bit words. However. this

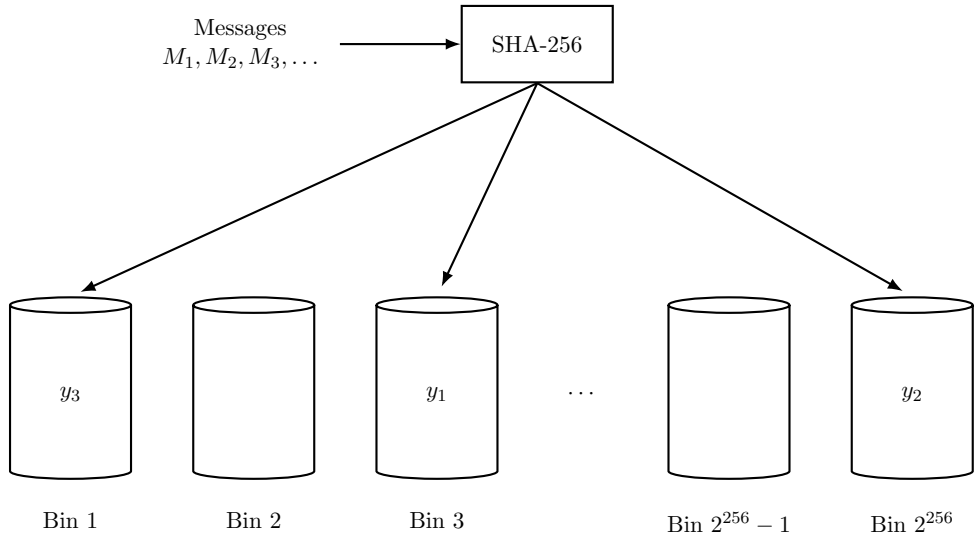


Figure 3.1: The SHA-256 hash function as bin assignment.

complicated structure makes finding preimages and collisions difficult. To see this, let us think about the hypothetical steps involved in finding a preimage of a 256-bit message digest y. Let us assume a message length of 100 bits and represent the message M by the variables $m_0 m_1 \cdots m_{99}$. Finding a 100-bit preimage of y is equivalent to solving for the variables m_i such that the SHA-256 hash of M is y. Note that there may be no 100-bit message whose SHA-256 hash is y, as $2^{100} < 2^{256}$. To avoid this case, let us assume that y was obtained by hashing a 100-bit message. So there exists at least one 100-bit preimage. Padding the message M will result in one 512-bit message block $M^{(1)}$ given by

$$M^{(1)} = \underbrace{m_0 m_1 m_2 \cdots m_{98} m_{99}}_{M}\ 1\ \underbrace{00000 \cdots 00000}_{347\text{ zeros}}\ \underbrace{00 \cdots 01100100}_{M\text{'s length in binary}}\ .$$

To solve for the variables m_i, we have to solve the equation $f(M^{(1)}, H^{(0)}) = y$. We would have to express each bit in the 256-bit message digest y as a function of the variables m_i and find a simultaneous solution to the resulting 256 equations. While it is easy to calculate the compression function output for a given input, it is difficult to express the output bits as a function of the input. Even if we somehow managed to find the 256 equations, they will involve complicated functions of the variables and solving them simultaneously is still difficult.

To see why finding second preimages or collisions for SHA-256 is hard, let us interpret each of the 2^{256} message digests as corresponding to a bin. Each input message is assigned to one of these bins in a deterministic manner (see Figure 3.1). But one cannot predict which bin a particular message will go

to without performing the hash calculation. Even if messages differ in only one bit, their corresponding bin assignments are very different. For instance, the SHA-256 message digests y_1 and y_2 of the 4-bit messages $M_1 = 1000$ and $M_2 = 1001$ are given in hexadecimal format by

$$y_1 = \texttt{40510175 845988F1 3F6162ED 8526F0B09F733844 67FA855E 1E79B44A 56562A58},$$
$$y_2 = \texttt{FE675FE7 AAEE830B 6FED09B6 4E034F84DCBDAEB4 29D9CCCD 4EBB90E1 5AF8DD71}.$$

Finding a second preimage is equivalent to fixing one of the bins and finding a message which maps to that bin. Finding a collision is equivalent to finding two distinct messages which are assigned to the same bin. But the number of bins is 2^{256} which is very large. Finding a second message which maps to a fixed bin or a pair of messages which map to the same bin is extremely unlikely.

To get an idea of how unlikely such events are, let us asssume that messages are equally likely to be assigned to any of the 2^{256} bins. Then the probability of finding a second preimage of a given message using n distinct messages is approximately $\frac{n}{2^{256}}$. The number of messages n would have to be close to 2^{256} to make this probability non-negligible. The probability of finding a collision using n distinct messages is approximately $\frac{n^2}{2^{257}}$. In this case, n would have to be close to 2^{128} to make this probability non-negligible.

Let us now discuss the motivation behind the specific message padding scheme used in SHA-256. The main requirement for a padding scheme used in a hash function is that distinct messages should not result in the same padded message. Let $\text{pad}(M)$ be the result of padding a message M. If $M \neq M'$ but $\text{pad}(M) = \text{pad}(M')$, then M and M' will have the same message digest resulting in a collision. For example, consider the simple padding scheme which consists of only appending zeros to a message M until the length becomes a multiple of 512. Then the messages 111 and 1110 will result in the same padded message which consists of 3 ones followed by 519 zeros.

The padding scheme used in SHA-256 appends a 1 followed by zeros and the length of the message as a 64-bit field. Suppose that the length field was not appended and the number of zeros appended after the 1 was chosen to make the padded message length a multiple of 512. This padding scheme satisfies $\text{pad}(M) \neq \text{pad}(M')$ for all $M \neq M'$. To see this, first consider the case when messages M and M' have the same length. Then the same bit string $S = 100\cdots00$ will be appended to both of them during the padding operation resulting in $\text{pad}(M) = M\|S$ and $\text{pad}(M') = M'\|S$ where $\|$ denotes the concatenation operation between bit strings. Clearly, $\text{pad}(M) \neq \text{pad}(M')$ if $M \neq M'$. Now consider the case when M and M' have different lengths. Let $\text{pad}(M) = M\|S$ and $\text{pad}(M') = M'\|S'$ where both S and S' have the

form $100\cdots00$ with a different number of zeros after the 1. Then $\text{pad}(M) \neq \text{pad}(M')$ since they both end in a different number of zeros.

If the length field is not required to guarantee that distinct messages result in distinct padded messages, then why is it appended to the message? Appending the length field prevents some types of attacks for finding collisions or second preimages. We will not discuss these attacks here and refer the reader to Chapter 9 of the *Handbook of Applied Cryptography* for more details[2].

Note that if the length field is finally appended to the message, we could technically avoid appending the single 1 and only append zeros. This padding scheme would also result in distinct padded messages for distinct messages. The number of 512-bit blocks generated by appending only zeros is sometimes smaller than the number of 512-bit blocks generated by appending a 1 followed by zeros. For example, suppose the message M is 448 bits long. The padding scheme which appends only zeros will append the 64-bit length field L resulting in the single 512-bit block $M\|L$. The padding scheme which appends a 1 followed by zeros cannot accommodate the length field in the 63 bits which remain in the first block after the 1 is appended to M. So it adds 511 zeros after the 1 increasing the intermediate padded message length to 960 bits. Appending the 64-bit length field L gives a 1024-bit final padded message $M\|1\|000\cdots000\|L$ which splits into two 512-bit blocks. In spite of this disadvantage, the padding scheme which appends a 1 followed by zeros and a length field has been used in SHA-256. This was probably a result of a conservative design approach taken by the SHA-256 designers.

3.2 RIPEMD-160

The RIPEMD-160 hash function was announced in 1996 by researchers from the German Information Security Agency and the Katholieke Universiteit Leuven, Belgium. It is an enhanced version of an earlier hash function called RIPEMD which was developed in 1992 as part of the European Union project RACE Integrity Primitives Evaluation (RIPE). The MD in RIPEMD stands for "message digest". The number 160 indicates the output length in bits.

As in SHA-256, the input to RIPEMD-160 can be at most $2^{64} - 1$ bits long. The padding scheme is similar to the one used in SHA-256 with a single 1 being appended to the message followed by some zeros and a 64-bit field which contains the message length. The number of zeros appended is chosen to make the padded message length a multiple of 512. The difference is that the 64-bit length field appears in little-endian format, i.e. the least significant 32-bit word appears first. For example, message length of 6 bits would be appended as `0x0000 0000 0000 0006` in the SHA-256 padding scheme and as `0x0000 0006 0000 0000` in the RIPEMD-160 padding scheme.

[2]Available for free download at `http://cacr.uwaterloo.ca/hac/`

The hash computation is an iterative process like in SHA-256 but the chaining variable used is 160 bits long and the compression function is different. Let the padded message consist of N 512-bit blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. The state of the computation is initialized by setting the 160-bit initial hash value $H^{(0)}$ to a fixed constant. The ith message block $M^{(i)}$ is combined with the previous hash value or chaining variable $H^{(i-1)}$ using a compression function g to generate the next hash value $H^{(i)}$.

$$H^{(i)} = g(M^{(i)}, H^{(i-1)}), \quad 1 \leq i \leq N. \tag{3.5}$$

Here the function g accepts 672 bits as input and returns 160 bits as output. The final hash value $H^{(N)}$ corresponds to the 160-bit output of the RIPEMD-160 hash function.

All that remains to completely specify the RIPEMD-160 hash function is to define the compression function g as we had defined the compression function f used in SHA-256. We will not define g here and refer the reader to Algorithm 9.55 of the *Handbook of Applied Cryptography* for an exact description. The description of the SHA-256 compression function f in the previous section served as a concrete example of a complicated input-output relationship which is easy to compute in the forward direction but resistant to finding preimages, second preimages, and collisions. It would be redundant to give yet another example of the same phenomenon by describing g.

3.3 P2PKH Addresses

We now describe an instance of how cryptographic hash functions are used in the Bitcoin system design. The most common operation in the Bitcoin system is the transfer of bitcoins between entities. In order to transfer bitcoins to a receiver, the sender needs to know the receiver's Bitcoin address. In this section, we will describe one type of Bitcoin address called the *Pay To Public Key Hash (P2PKH) address*.

P2PKH addresses are derived from a ECDSA public key using the SHA-256 and RIPEMD-160 hash functions. Before considering this derivation, let us discuss the Base58 encoding which is a method to represent byte strings using alphanumeric characters. P2PKH addresses are finally represented as Base58-encoded strings. From the 62 alphanumeric characters (10 numeric digits, 26 uppercase letters, 26 lowercase letters), the number 0, the uppercase letter O, the uppercase letter I, and the lowercase letter l are excluded to avoid confusion and errors while reading P2PKH addresses. The remaining 58 alphanumeric characters represent the integers from 0 to 57 as shown in Table 3.1. Given a byte string $b_n b_{n-1} \cdots b_0$, the Base58 encoding represents it using the 58 alphanumeric characters as follows:

Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int
1	0	A	9	K	18	U	27	d	36	m	44	v	53
2	1	B	10	L	19	V	28	e	37	n	45	w	54
3	2	C	11	M	20	W	29	f	38	o	46	x	55
4	3	D	12	N	21	X	30	g	39	p	47	y	56
5	4	E	13	P	22	Y	31	h	40	q	48	z	57
6	5	F	14	Q	23	Z	32	i	41	r	49		
7	6	G	15	R	24	a	33	j	42	s	50		
8	7	H	16	S	25	b	34	k	43	t	51		
9	8	J	17	T	26	c	35	l	44	u	52		

Table 3.1: Base58 character (Ch) to integer (Int) mapping

1. Encode each leading zero byte (if any) as a 1. For example, if the first m leading bytes b_n, b_{n-1}, ..., b_{n-m+1} are all zero bytes then they will be encoded as m ones.

2. Let b_{n-m} be the first byte which is not a zero. Let N be the integer whose big-endian representation is given by $b_{n-m}b_{n-m-1}\cdots b_0$. Then representing each byte b_i as an integer from 0 to 255, we have

$$N = \sum_{i=0}^{n-m} b_i 256^i$$

3. Convert N to a base 58 representation $a_k a_{k-1}\cdots a_0$ where each a_i is an integer from 0 to 57 and

$$N = \sum_{i=0}^{k} a_i 58^i.$$

4. Map each integer in the sequence $a_k a_{k-1}\cdots a_0$ to the corresponding Base58 character in Table 3.1 and append the resulting string to the m ones from step 1.

For example, consider the byte string `0x00001234` given in hexadecimal format. In base 256, it is given by `0 0 18 52`. It has two leading zero bytes and the remaining portion corresponds to the integer $N = 4660 = 18 \times 256 + 52$. In base 58, N is given by `1 22 20` as $4660 = 58^2 + 22 \times 58 + 20$. The Base58 encoding of the byte string `0x00001234` is then given by `112PM` where the two leading ones encode the two leading zero bytes, the `2` encodes the integer `1`, `P` encodes `22` and `M` encodes `20`.

Recall that a ECDSA public key in Bitcoin is a point kP on the `secp256k1` elliptic curve where k is a 256-bit integer representing the private key and P is the base point. In uncompressed format, the public key can be represented using 65 bytes consisting of the single byte `0x04` followed by the 32-byte x

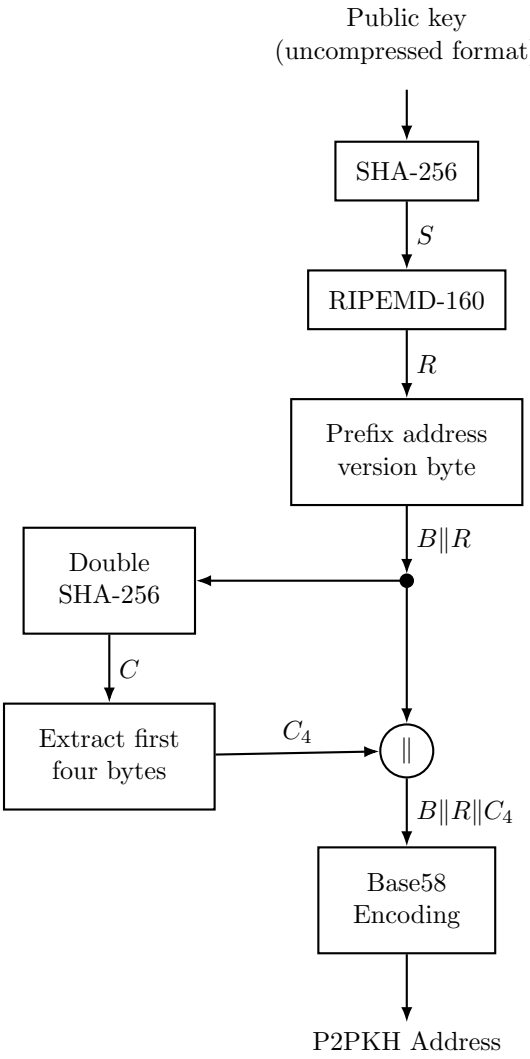


Figure 3.2: Generating a P2PKH address from a public key

and y coordinates of kP (see Section 2.4). Let X and Y denote the 32-byte big-endian representations of the x and y coordinates of kP respectively. Let $\|$ denote the bit string concatenation operator. The P2PKH address generation procedure is illustrated in Figure 3.2 and proceeds as follows:

1. Calculate the SHA-256 hash of the uncompressed public key $\texttt{0x04}\|X\|Y$:

$$S = \text{SHA-256}\left(\texttt{0x04}\|X\|Y\right).$$

   S is 32 bytes long.

2. Calculate the RIPEMD-160 hash of S:

$$R = \text{RIPEMD-160}(S).$$

   R is 20 bytes long.

3. Prefix R with a single byte B which contains the address version number to get $B\|R$ which is 21 bytes long. The address version number for P2PKH addresses is `0x00` on the main Bitcoin P2P network and `0x6f` on *testnet* which is a network used for testing Bitcoin features.

4. Calculate the result of applying the SHA-256 hash function twice on $B\|R$:

$$C = \text{SHA-256}\left(\text{SHA-256}\left(B\|R\right)\right).$$

   C is 32 bytes long.

5. Let C_4 denote the first four bytes of C. Append C_4 to $B\|R$ to get a 25-byte string $B\|R\|C_4$.

6. Encode $B\|R\|C_4$ using Base58 encoding to the get the P2PKH address A:

$$A = \text{Base58}(B\|R\|C_4).$$

When the address version byte B is `0x00`, the P2PKH address begins with a `1` as the Base58 encoding represents leading zero bytes with ones. For example, the P2PKH address corresponding to the public key equal to the base point P given in equation (2.4) is given by `1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm`. When B is `0x00`, the P2PKH address consists of at most 34 Base58 characters including the leading `1`. This is because the largest base 256 integer which can fit in the 24-byte $R\|C_4$ field is $256^{24}-1$. This number is smaller than $58^{33}-1$, the largest base 58 number which has 33 digits. The P2PKH address can have less than 34 Base58 characters because the integer in $R\|C_4$ can sometimes be represented using 32 digits in base 58. While typing a Base58-encoded P2PKH address is cumbersome, it is more convenient than typing the 25-byte $B\|R\|C_4$ as 50 hexadecimal digits or 200 bits.

The C_4 field serves the role of a checksum which can be used to detect errors introduced while typing a P2PKH address. Before an address A is used, Base58 decoding is performed to get $B\|R\|C_4$. The double SHA-256 hash of $B\|R$ is calculated and checked to be equal to C_4. This is called checksum validation. The field C_4, being the partial output of the double SHA-256 function, can be assumed to behave like a random 32-bit value. If someone were to make an error in typing or writing down the P2PKH address A, then the checksum validation will succeed only with a probability $\frac{1}{2^{32}}$. *But why invoke the SHA-256 function twice when one invocation would also yield a checksum with the same property?* The double SHA-256 function

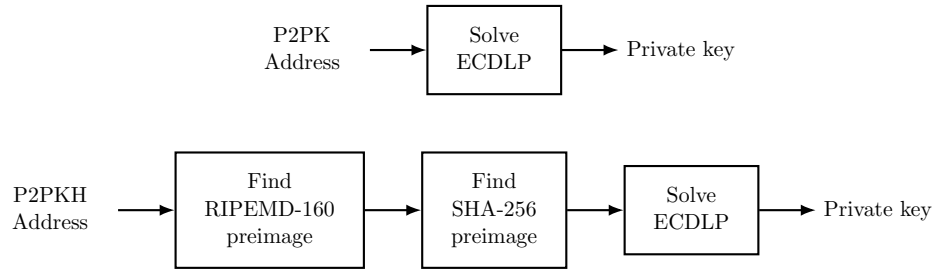


Figure 3.3: Steps involved in recovering a private key from a P2PK address and a P2PKH address

is resistant to length-extension attacks which affect the SHA-256 function. In a length-extension attack, knowing only hash SHA-256(M) but not the message M still allows an attacker to calculate the hash SHA-256($M\|M'$) of some extended message $M\|M'$. In order to be resistant to length-extension attacks, the double SHA-256 function is used to calculate the message digest in the ECDSA (see Section 2.5). While there is no known way of using this attack to compromise the Bitcoin system if a single SHA-256 invocation was used to generate the checksum, the use of the double SHA-256 function is probably just a conservative design choice.

But why is the public key hashed in the first place? Why not use the public key itself or a Base58 encoding of it as the Bitcoin address? The Bitcoin system does allow the use of a public key to identify the receiver of some bitcoins. In this case, the public key of the receiver is called a *Pay-to-Public-Key (P2PK) address*. But P2PKH addresses are preferred over P2PK addresses for security reasons. To understand the security risk in using P2PK addresses, suppose some bitcoins are transferred to a receiver's Bitcoin P2PK address which is equal to her public key kP. To spend these bitcoins later, the receiver has to create a digital signature using the private key k. This is equivalent to proving knowledge of k without actually revealing it. Anyone with knowledge of k effectively owns the bitcoins deposited in the Bitcoin address derived from kP. So if P2PK addresses are used, then an adversary can attempt to calculate k from kP. While there are no efficient methods currently known for solving this elliptic curve discrete log problem (ECDLP), it is possible that such methods may be discovered in the future. By hashing the public key to derive the P2PKH address from kP, an adversary looking to derive the private key k from the P2PKH address has to solve the additional difficult problem of finding the preimage of a hash function output before attempting a solution to the ECDLP. So hashing the public key effectively amounts to hiding it from adversaries who may in the future be capable of solving the ECDLP. If only one hash function, say RIPEMD-160, had been used to hash the public key, then the adversary would have to find preimages for RIPEMD-160 outputs. By hashing the public key with both SHA-256

and RIPEMD-160, an adversary is required to find preimages for both these functions, which is a harder problem. Figure 3.3 shows the steps involved in recovering a private key from a P2PK and a P2PKH address. We have not shown the Base58 decoding step which needs to be performed on the P2PKH address before attempting to find the RIPEMD-160 preimage because it is trivial to perform. From the perspective of an adversary who is trying to recover the private key from a P2PKH address, the order in which the two hash functions are applied to the public key does not matter. But if RIPEMD-160 had been applied on the public key first followed by SHA-256, then the resulting hash value would be 32 bytes instead of 20 bytes and the P2PKH address would be longer.

Chapter 4

The Blockchain

The blockchain is the database containing a record of all Bitcoin transactions since Bitcoin came into existence in 2009. The blockchain consists of a linear list of *blocks* where each block is composed of a block header followed by a list of Bitcoin transactions. This is illustrated in Figure 4.1. A Bitcoin transaction involves the transfer of bitcoins between entities. We will specify the format of a transaction in the next chapter. For now, a transaction can be thought of as an encoding of the details of a transfer of bitcoins from source Bitcoin addresses to destination Bitcoin addresses.

The first block in the blockchain (the *genesis block*) was created in January 2009. As of July 2017, the blockchain had more than 478,000 blocks and occupied approximately 125 gigabytes of disk space. Until August 2017, the maximum size of a block was 1 megabyte (1,000,000 bytes). In August 2017, a new feature called *Segregated Witness (SegWit)* was activated in the Bitcoin network which effectively increased the maximum block size to 4 megabytes.

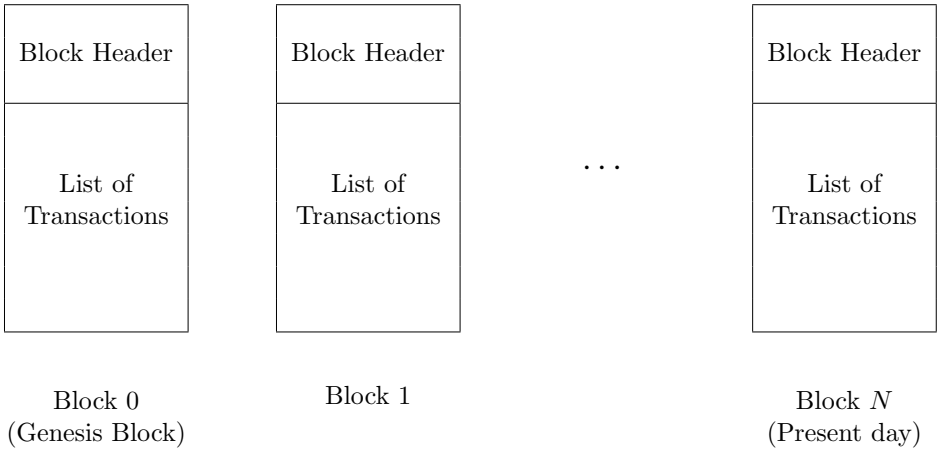


Figure 4.1: The blockchain

The motivation for and design of SegWit will be discussed in the next chapter as it requires us to first understand some shortcomings in the pre-SegWit Bitcoin system.

In this chapter, we will describe the blockchain and explain the motivation behind different aspects of its design.

4.1 Rewarding Blockchain Updation

The task of storing and updating the blockchain is performed collectively by the nodes in the Bitcoin P2P network. Nodes called *full nodes* store a copy of the blockchain on their hard disks. When a full node connects to the Bitcoin network for the first time, it downloads a copy of the blockchain from the existing full nodes. The task of adding blocks to the blockchain is called *mining* and is done by full nodes called *miners*. The naming convention for mining/miners was chosen because the mining task involves the solution to a difficult computational problem and a miner which successfully solves such a problem is rewarded with *newly created* bitcoins. This reward is called the *block subsidy* and is currently equal to 12.5 bitcoins per block. In addition to the block subsidy, miners also receive *transaction fees* in bitcoins which are provided by the transactions in the block being added. The sum of the block subsidy and the transaction fees is called the *block reward*.

Mining is the only way new bitcoins are created in the Bitcoin system. The computational difficulty of mining a single block is adjusted by the network to ensure that a new block is added approximately every 10 minutes. This schedule along with the size of the block subsidy controls the rate of new bitcoin creation. The block subsidy was 50 bitcoins per block in 2009 when Bitcoin came into existence. It is halved every 210,000 blocks which is about four years assuming it takes 10 minutes to mine a new block. The block subsidy became 25 bitcoins in November 2012 when block 210,000 mined and 12.5 bitcoins in July 2016 when block 420,000 was mined. The smallest indivisible unit of the Bitcoin currency is called a *satoshi* with each bitcoin being equal to 100 million satoshi. As the block subsidy is progressively halved, it will eventually become less than 1 satoshi. At this point, it will be considered zero. The block subsidy will become zero when block 6,930,000 is mined which is expected to be around the year 2140. Once the block subsidy becomes zero, transaction fees will be the only incentive for miners to continue mining new blocks. As the rate of new bitcoin creation decreases geometrically, the total number of bitcoins which will ever come into existence is about 21 million. While there is nothing special about the specific constants chosen to represent the initial block subsidy and the halving schedule, the motivation behind having a fixed limit on the total number of bitcoins is to prevent inflation of the currency.

We will discuss the details of the computational problem used in Bitcoin mining in Section 4.3 after we describe the block header structure.

`nVersion`	4 bytes
`hashPrevBlock`	32 bytes
`hashMerkleRoot`	32 bytes
`nTime`	4 bytes
`nBits`	4 bytes
`nNonce`	4 bytes

Figure 4.2: The block header fields and their sizes

4.2 The Block Header

Every block in the blockchain begins with a 80-byte block header. The fields in the block header and their respective sizes are shown in Figure 4.2. The field names are taken from the source code of the Bitcoin Core reference client. The prefix `n` of the 4-byte fields is a convention to indicate that they are integer variables. The prefix `hash` of the 32-byte fields indicates that they store hash function outputs.

The block header begins with a 4-byte `nVersion` field which specifies the version of the block. As the Bitcoin system evolved, changes were proposed to fix bugs or enable new features. The version number of a block indicates which features are supported by the transactions present in it.

Each block is identified by the double SHA-256 hash of its block header. This is called the *block hash* of the block. The `hashPrevBlock` field in the block header contains the block hash of the previous block in the blockchain. This is illustrated in Figure 4.3. Since the genesis block has no previous block, its `hashPrevBlock` field was set to all zeros. The block headers of two distinct blocks will differ in at least in the `hashPrevBlock` field. Since the SHA-256 output behaves like a random 256-bit string, the probability that the block hashes of two distinct blocks will be the same is extremely negligible. So the block hash can be safely considered to be a unique identifier of a block.

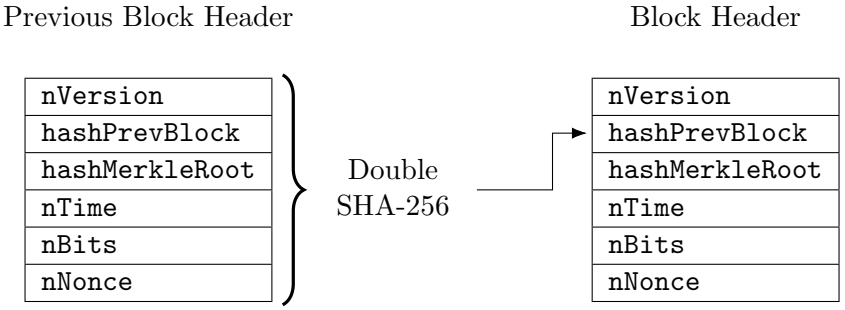


Figure 4.3: The `hashPrevBlock` field contains the double SHA-256 hash of the previous block header

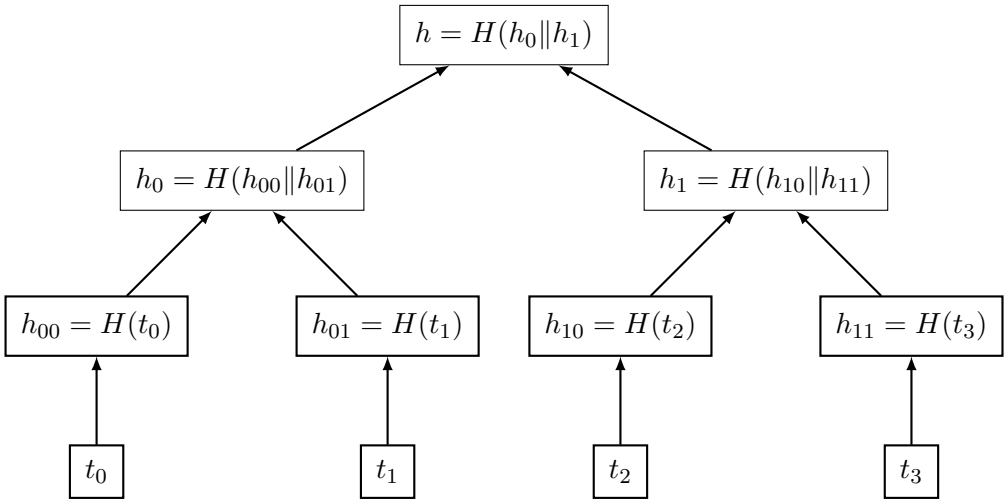


Figure 4.4: Merkle tree of four transactions

The `hashMerkleRoot` field stores the root hash of a Merkle tree formed using the transactions in the block. The transactions are arranged as a list and the double SHA-256 hash of each of them is computed. Using these hashes as leaves, a binary tree is created where each node is associated with a double SHA-256 hash of the concatenation of its child hashes. This is illustrated in Figure 4.4 for the case when four transactions are used to construct a Merkle tree. In the figure, t_0, t_1, t_2, t_3 represent transactions, $\|$ denotes the concatenation operator, and $H(\cdot)$ is used to denote the double SHA-256 function, i.e. $H(x) = \text{SHA-256}(\text{SHA-256}(x))$. The hash value h associated with the root of the tree is called the *root hash* or *Merkle root* of the tree. When the

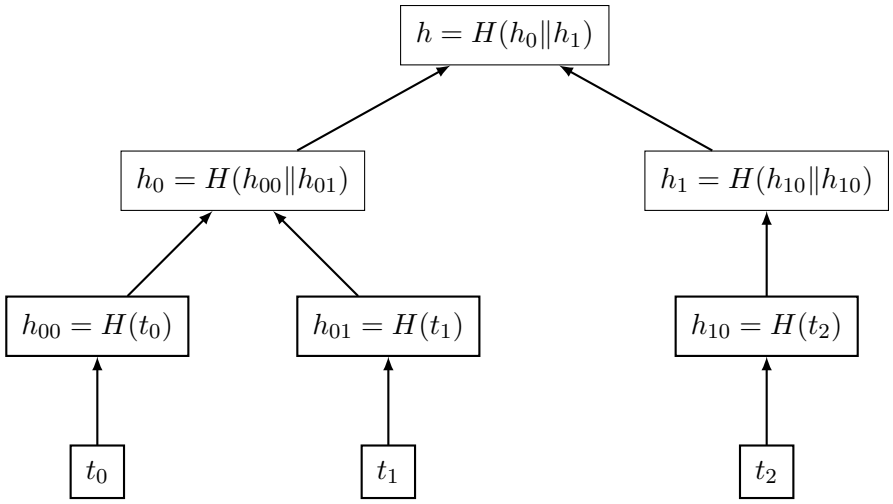


Figure 4.5: Merkle tree of three transactions

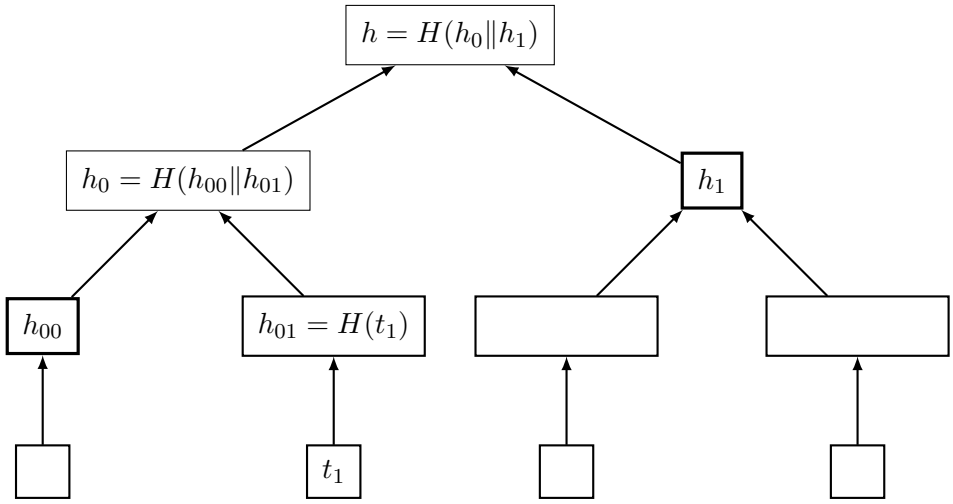


Figure 4.6: Proof of membership of transaction t_1 in a block with `hashMerkleRoot` equal to h

number of transactions is not a power of two, some nodes in the Merkle tree will have only one child. In that case, the hash of the single child is concatenated with itself and then hashed to derive the hash value associated with the parent node. Figure 4.5 illustrates the calculation of the Merkle root corresponding to three transactions. In this case, the node corresponding to h_1 has only one child h_{10}. The value h_{10} is concatenated with itself and hashed to get h_1.

The root hash stored in `hashMerkleRoot` is a compact representation of all the transactions in a block. Any change to a transaction in a block will result in a change in the `hashMerkleRoot` field as the SHA-256 function is collision resistant. A change in the `hashMerkleRoot` will in turn result in a change in the block hash of the block. The block hash of a block also depends on the `hashMerkleRoot` of the previous block through the `hashPrevBlock`. As this dependence is recursive, the block hash of a particular block depends on *all the transactions in all the previous blocks* all the way upto the genesis block. Changing any past transaction will involve a recalculation of all the block hashes of blocks which are subsequent to and including the block containing that transaction. This property will turn out to be important for guaranteeing tamper resistance of the transaction data.

But why use a Merkle tree of the transactions? Why not hash the concatenation of all the transactions and put the resulting hash in the block header? To guarantee tamper resistance of the transaction data, including the hash $H(t_0 \| t_1 \| \cdots \| t_{n-1})$ of the transactions $t_0, t_1, \ldots, t_{n-1}$ in the block header instead of the Merkle root would have been sufficient. The reason for using the Merkle root is that it enables efficient membership proofs of transactions within a block. For example, suppose we want to prove that the transaction t_1

was involved in the calculation of the root hash h in Figure 4.4. We only need to provide the *Merkle branch* consisting of the hashes h_{00} and h_1 as shown in Figure 4.6. The hashes h_{01}, h_0, and h can be calculated from t_1 and the Merkle branch. If the root hash h appears in the `hashMerkleRoot` field of a block, then by the second preimage resistance of the SHA-256 hash function we can be certain that this block contains the transaction t_1. In general, the Merkle branch required to prove the existence of a transaction in a block containing n transactions has size $O(\log_2 n)$. On the other hand, if $H(t_0\|t_1\|\cdots\|t_{n-1})$ had been used instead of the Merkle root then the membership proof of a transaction would require us to specify all the transactions in the block whose size is $O(n)$. There are nodes in the Bitcoin network called *simple payment verification (SPV) nodes* which store only the block headers and not the whole blocks like full nodes. When they require information about transactions in the blockchain which contain their Bitcoin addresses, they contact full nodes which respond with Merkle branches to prove the existence of the relevant transactions.

The last three fields in the block header `nTime`, `nBits`, and `nNonce` are related to mining. They are explained in the next section.

4.3 Mining

Mining is the process by which new blocks are added to the blockchain. Each block consists of a block header followed by a list of transactions. The list begins with a special transaction called the *coinbase transaction* which encodes the transfer of the block reward (block subsidy plus the transaction fees from the other transactions) to the miner which added the block to the blockchain. Each coinbase transaction involves the creation of new bitcoins. The amount of bitcoins created is equal to the block subsidy which is currently 12.5 bitcoins. The other transactions in the list are called *regular transactions*. They encode the transfer of bitcoins which were created in some previous block. A block must contain exactly one coinbase transaction but it may contain zero or more regular transactions. The maximum number of regular transactions in a block is limited by the block size which was 1 MB until August 2017 and 4 MB after that.

Nodes which want to record new regular transactions in the blockchain broadcast them on the Bitcoin network. When other nodes hear these new transactions, they add them to a transaction *memory pool (mempool)* which is stored in local memory (RAM). A miner node forms a candidate block by collecting some transactions from its mempool. The miner includes a coinbase transaction in the candidate block which transfers the block reward to its own Bitcoin address. There will be several miner nodes competing to add the next block in the blockchain and claim the resulting block reward. The candidate blocks created by these different miner nodes will differ in the coinbase trans-

actions as each node will insert its own Bitcoin address as the recipient of the block reward. The candidate blocks may also differ in the regular transactions included in them as different miner nodes may have different sets of transactions in their respective mempools. This may be due to the miner nodes receiving transactions broadcasted on the Bitcoin network at different times due to network latencies.

The *height* of a block in the blockchain is the number of blocks preceding it. The genesis block has height 0, the immediate successor of the genesis block has height 1 and so on. Suppose a miner node is attempting to add a candidate block at height N. The newest block in the node's copy of the blockchain has height $N-1$. The `hashPrevBlock` field of the candidate block header is populated with the block hash of the block at height $N-1$. The `hashMerkleRoot` field is populated with the Merkle root of the transactions in the candidate block. The `nVersion` field contains the current block version number.

The `nTime` field is populated with a timestamp in Unix time format to record the time of candidate block creation. The Unix time is the number of seconds which have elapsed since 12:00 AM Coordinated Universal Time on January 1st, 1970 with deductions to account for leap seconds.[1] Each node in the network has a local clock which is not necessarily synchronized with the local clocks of the other nodes. So there is no globally unique notion of time in the network. The Bitcoin system does not specify an explicit algorithm for calculating the `nTime` field in a candidate block. However, it imposes two constraints to ensure that the timestamp in the `nTime` field is approximately correct:

- In a candidate block at height N, the `nTime` field is required to be strictly greater than the median of the `nTime` values in the 11 blocks in the blockchain at heights $N-1, N-2, \ldots, N-11$. This median value is called the *median-time-past* of the block at height $N-1$. Note that this constraint causes the median-time-past values to increase monotonically with block height, even if the `nTime` fields do not.

- When a network node receives a candidate block created by a miner, it rejects it if the `nTime` field specifies a time which exceeds the node's *network-adjusted time* by more than two hours. The network-adjusted time at a node is the median of the local clocks of the other nodes it is connected to.

A miner node is free set to the `nTime` field to any value which satisfies these constraints. The first constraint specifies a lower bound on `nTime` which can be calculated from the current blocks in the blockchain. The upper bound specified by the second constraint cannot be explicitly calculated by the miner

[1] See `https://en.wikipedia.org/wiki/Unix_time`

nBits	Target Threshold	$b_1 - 3$
0x03123456	0x123456	0
0x02123456	0x1234	-1
0x05123456	0x1234560000	2
0x08123456	0x1234560000000000	5

Table 4.1: Examples of `nBits` field values and corresponding target thresholds

as it does not know the network-adjusted times of the other nodes in the network. But it can hope to satisfy the upper bound by using `nTime` values which are equal or close to its own network-adjusted time.

The `nBits` field in the block header encodes a 256-bit unsigned integer called the *target threshold* using a base 256 version of the scientific notation. Let $b_1b_2b_3b_4$ be the four bytes in `nBits`. The first byte b_1 plays the role of the exponent and the remaining three bytes encode the mantissa. The target threshold T is derived as

$$T = b_2b_3b_4 \times 256^{b_1-3},$$

where b_1 and $b_2b_3b_4$ are interpreted as unsigned integers. Some examples of `nBits` field values and the corresponding target thresholds are given in Table 4.1. As illustrated by the examples, the mantissa is shifted to the left by $b_1 - 3$ bytes. The value of b_1 is never allowed to be larger than 34 to ensure that the target threshold can be represented by a 32-byte (256-bit) string. The target threshold is a network-wide setting which is adjusted by all the network nodes every 2,016 blocks.

Recall that the block hash of a block is the double SHA-256 hash of its block header. *The goal of the miner is to find a candidate block whose block hash is less than or equal to the target threshold.* The miner is free to set the value of the `nNonce` field to any value in order to find such a block hash. The word nonce is used in cryptography to denote a number which is used only once. Since the SHA-256 function is preimage resistant, the only thing a miner can do is to try different values for the `nNonce` field and check each time if the block hash falls below the threshold. Assuming that the output of the SHA-256 function behaves like a random 256-bit string where each bit is equally likely to be 0 or 1 independently of the other bits, the probability that the block hash falls below the target threshold T for a trial `nNonce` value is

$$p = \frac{T+1}{2^{256}}.$$

Modelling each try by the miner as a Bernoulli random process with success probability p, the average number of trials required to find a block hash below the threshold is $1/p$. For example, the `nBits` field on January 1, 2017 was `0x180375ff` which corresponds to an exponent of 24 and a mantissa of

$226815 \approx 262144 = 2^{18}$. In this case, the average number of trials required was approximately

$$\frac{2^{256}}{T+1} = \frac{2^{256}}{226815 \times 256^{24-3}+1} \approx \frac{2^{256}}{2^{18} \times 2^{168}} = 2^{70}.$$

Such a large number of trials required to find a valid block hash is the reason why mining is a computationally difficult problem. A miner which successfully finds a block hash for a candidate block below the target threshold is said to have *found or mined a valid block.* Since the only way to mine a valid block is to search through a large number of candidate block hashes, the valid block is called a *proof-of-work (PoW) solution.* It proves that a certain amount of work was performed on the average in order to find it.

What happens after a miner finds a valid block at height N? Such a miner immediately broadcasts the block on the Bitcoin network. It also appends the block to its local copy of the blockchain and begins mining for the next block at height $N+1$. When other nodes receive this broadcasted block, their reaction depends on the state of their local copy of the blockchain. For now, assume that all the other nodes in the network have the same copy of the blockchain consisting of blocks from the genesis block to a block at height $N-1$. We will relax this assumption later. When the new block at height N arrives, miner nodes which are still mining their candidate blocks at height N stop mining, append the new block to their local copy of the blockchain, and start mining for the next block at height $N+1$. If the receiving node is a full node which does not perform mining, then it will just add the new block to its local copy of the blockchain.

How is the target threshold value chosen? The rate at which a computing device can calculate block hashes is measured in megahashes per second (MH/s), gigahashes per second (GH/s), or terahashes per second (TH/s). These units correspond to 10^6, 10^9, and 10^{12} hashes per second respectively. A typical personal computer (PC) can calculate block hashes at a rate less than 100 MH/s. To calculate 2^{70} block hashes, a PC operating at 100 MH/s will require more than 300,000 years. Nowadays, mining is done using application specific integrated circuits (ASICs) designed specifically to compute several instances of the double SHA-256 function in parallel. One can purchase mining rigs which combines several such ASIC chips to deliver hash rates of the order of a few TH/s. A single mining rig operating at 1 TH/s will still require more than 30 years to calculate 2^{70} hashes. The mining landscape is dominated by companies which have consolidated thousands of such mining rigs into datacenters in locations with low electricity and cooling costs. There are also *mining pools* where geographically distributed nodes combine their respective mining hash rates to reduce the time required to mine a valid block. The total hash rate available across the whole Bitcoin network on January 1, 2017 was estimated[2] to be 2,463,610 TH/s. Using this hash rate, 2^{70} block

[2]Source: `https://blockchain.info/charts/hash-rate`

hashes can be calculated in 8 minutes.

The Bitcoin protocol specifies that the average time required to mine a valid block should be 10 minutes. If the total hash rate R_{total} of all the miners in the Bitcoin network were known, then we know that the network can calculate $600R_{\text{total}}$ hashes in 10 minutes. The target threshold T could then be chosen such that the average number of trials required to mine a valid block is equal to $600R_{\text{total}}$, i.e. by solving for T in

$$\frac{2^{256}}{T+1} = 600R_{\text{total}}.$$

But the protocol does not have any means to directly measure R_{total}. Instead, the time which was spent in finding the previous 2,016 blocks is measured by taking the difference of the `nTime` fields of blocks whose heights differ by 2,016. Ideally, this time should be 20,160 minutes. If the actual time spent is less than this value, then the target threshold is decreased to decrease the probability of finding a valid block in a single trial. If the actual time spent is more than this value, then the target threshold is increased. The update formula for transforming the old target threshold T_{old} to a new value T_{new} is given by

$$T_{\text{new}} = T_{\text{old}} \times \frac{\text{Measured duration for finding 2,016 blocks in seconds}}{2016 \times 600}.$$

The target threshold is updated once every 2,016 blocks by all the nodes in the Bitcoin network. The number of 2,016 was chosen because it represents the number of blocks which would be found in two weeks if a block was found every 10 minutes, i.e. $2016 = 14 \times 24 \times 6$.

The Bitcoin protocol has a bug in the calculation used to measure the duration for finding 2,016 blocks. It was present in the original implementation of the Bitcoin client and is difficult to fix as it requires a *hard fork* change to the protocol[3]. Suppose a miner node is in the process of creating a candidate block whose height is a multiple of 2,016, say $2016n$. It measures the duration required to find the previous 2,016 blocks as the difference in the timestamp (`nTime`) fields of the blocks at height $2016n - 1$ and $2016(n - 1)$. So for $n = 1$, the difference in the timestamp values of the blocks at height 2,015 and 0 were used. Note that this duration actually measures the time required to find 2,015 blocks because the timestamps are inserted before the mining begins and the difference between the timestamps of blocks at height i and $i+1$ gives an estimate of the time required to find block i. The time required to find the block with height $2016n-1$ will require the timestamp of the block with height $2016n$. The effect of this bug is that the measured duration is reduced by about 10 minutes which causes the T_{new} value to be reduced by $T_{\text{old}}/2016$. This represents a change in the probability of finding a valid block of 0.05% which is negligible.

[3]Hard and soft fork protocol changes are discussed in Chapter 7

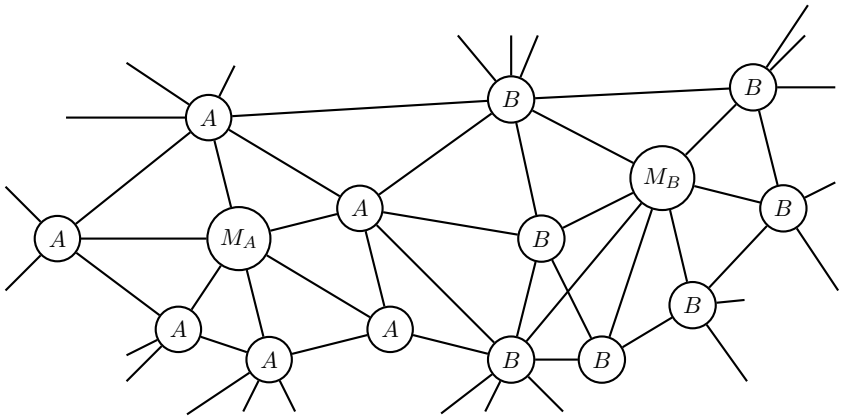
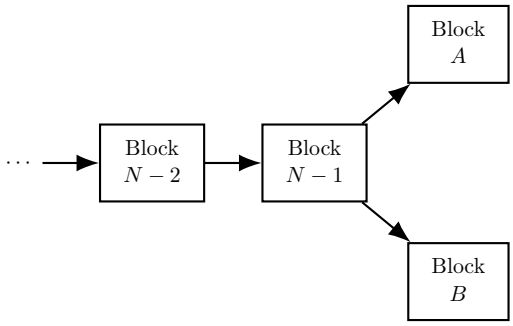


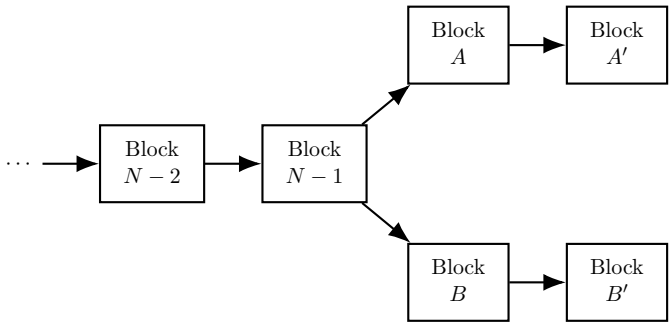
Figure 4.7: Illustration of network state in the event of a blockchain fork with two branches each having one block.

What if the miner runs out of `nNonce` *values to try?* Since the `nNonce` field is only 4 bytes long, a miner can generate only 2^{32} trials by modifying it. In case a miner cannot a find a block hash below the threshold in these trials, it can modify a field called the *coinbase* which is present in the coinbase transaction. This field is of variable length and can have a maximum length of 100 bytes. Except for the first four bytes which are reserved for a specific purpose, the coinbase can hold arbitrary data. Once a miner has exhausted the 2^{32} trials by modifying the `nNonce`, it can change some bits in the coinbase which will in turn modify the `hashMerkleRoot` value in the block header. The miner can now retry the 2^{32} `nNonce` values as they will result in new block hash values.

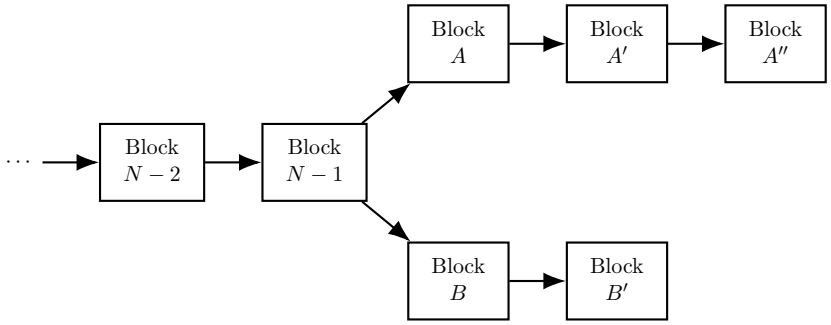
What if two miners find valid blocks at around the same time and broadcast the blocks? Once again let us assume that all the network nodes have the same copy of the blockchain which ends in a block at height $N-1$. Suppose two valid blocks A and B both at height N are found by two different miners and are broadcasted before each miner received the other miner's block. This is posssible due to the delays inherent in propagating blocks over the network. The other nodes either receive block A first or block B first. Each node will accept the first block at height N that it receives and reject the second block. So if a miner receives block A first it will append it to its local copy of the blockchain and start mining for a block at height $N+1$ with block A as the previous block. If the miner later receives block B, it will reject it. Eventually every node in the network would have received either block A or block B and extended its local copy of blockchain with the received block. This is illustrated in Figure 4.7 where M_A and M_B denote the miner nodes which created blocks A and B respectively. Edges have been drawn between nodes which are peers in the Bitcoin P2P network. Nodes labelled A have



(a) Block chain state in the event of a fork at height N



(b) Block chain state in the event that both branches in a fork get extended to equal height



(c) Block chain state in the event one branch in a fork becomes longer than the other

Figure 4.8: Block chain forks

received block A first and nodes labelled B received block B first. Such a situation is called a *blockchain fork* since the state of blockchain as seen by the network as a whole consists of two branches both originating from the same parent block at height $N - 1$.

Another way to represent the blockchain fork is shown in Figure 4.8(a). Nodes which first received block A extended their copy of the blockchain using the upper branch containing block A. The nodes which first received block B extended the blockchain using the lower branch. Both branches will have

some proportion of the miners in the Bitcoin network working to extend them. It is possible that valid blocks are found once again around the same time on both branches and broadcast on the network. This results in a situation shown in 4.8(b). Blocks A' and B' were found by miners trying to extend the branches containing blocks A and B respectively. Due to the randomness inherent in the mining process and the block propagation in the network, it is unlikely that both branches in a blockchain fork get extended to equal height indefinitely. Eventually, one branch will become longer than the other. This situation is shown in Figure 4.8(c) where the branch starting from block A has been extended to height $N + 2$ while the branch starting from block B has been extended to height $N + 1$. *The Bitcoin protocol requires the network nodes to switch to the longest branch they become aware of.* So when the block A'' is received by the miner nodes which are working on extending the branch starting at block B, they will switch to the branch starting at block A and begin mining candidate blocks which have block A'' as their previous block. They will request the intermediate blocks A and A' from the peer who communicated block A'' to them. Non-miner full nodes which have block B' as the latest block in their copy of the blockchain also switch to the branch starting from block A upon receiving block A''. The branch consisting of blocks B and B' will no longer be extended. Blocks belonging to such abandoned branches are called *stale blocks* and they are eventually deleted. By having all nodes switch to the longest branch, the protocol ensures that only a single linear list of blocks survives after the resolution of blockchain forks. The network is said to have achieved *consensus* about which linear list of blocks constitute the blockchain.

What about the transactions in the stale blocks? A transaction is valid only if it belongs to a block which survives after any blockchain forks have been resolved. The coinbase transactions in stale blocks become invalid. A regular transaction in a stale block could already be present in one of the blocks which survived after fork resolution. If not, it is added back to the mempool of transactions which nodes use to construct new candidate blocks.

4.4 Bitcoin Transactions

In this section, we give a high-level description of Bitcoin transactions in order to discuss the security properties of the blockchain. A more detailed description of the transaction format will be given in Chapter 5.

A Bitcoin transaction encodes a transfer of bitcoins between entities. A destination of the transfer in a transaction is called an *output*. A single transaction can have several outputs. Each output in a transaction can serve as a source of bitcoins in a later transaction. When previous transaction outputs are specified as sources of bitcoins in a transaction, they are called *inputs*. A coinbase transaction has no input and at least one output. There is no

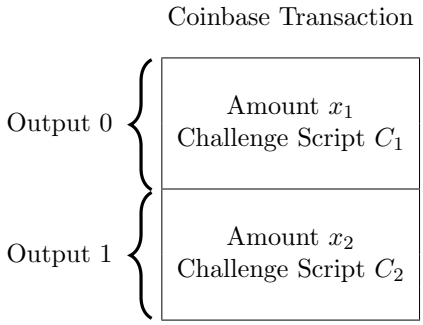


Figure 4.9: Illustration of a coinbase transaction.

input because the source of bitcoins is not a previous transaction output but the block reward, i.e. the sum of the block subsidy and the transaction fees from the transactions in the block. Each output in the coinbase transaction specifies two items:

- The amount of bitcoins from the block reward which are associated with this output.
- A script which specifies the conditions under which the bitcoins associated with this output can be spent.

The script in an output can be thought of as a challenge. An entity which provides a satisfactory response can transfer the bitcoins associated with the output. Figure 4.9 illustrates a coinbase transaction with two outputs. The first output specifies an amount x_1 and a challenge script C_1. A satisfactory response to C_1 is needed to spend the x_1 bitcoins. Similarly, a satisfactory response to the challenge script C_2 is needed to spend the x_2 bitcoins in the second output.

To see an example of a challenge and a satisfactory response to it, consider a miner who creates a block and wants the block reward to be paid to P2PKH addresses it owns. Ownership of a P2PKH address is the same as knowing the private key corresponding to the public key used to create the address. In this case, the challenge script in an output of the coinbase transaction will contain a P2PKH address. The challenge script will require anyone who wants to spend the bitcoins to provide a response script which consists of two items:

- A public key which hashes to the given P2PKH address.
- A digital signature created using the private key corresponding to the public key. The signature can be verified using the provided public key.

While a single output in the coinbase transaction is sufficient for a miner to gain control of the block reward, multiple outputs give the miner flexibility to distribute the block reward to multiple addresses.

The sum of the amounts in all the outputs of the coinbase transaction should not exceed the block reward. Suppose that the block reward in the block containing the coinbase transaction of Figure 4.9 is R bitcoins. Then the amounts x_1 and x_2 must satisfy $x_1+x_2 \leq R$. This ensures that the transaction does not spend more than the amount of bitcoins which are available for spending. If $x_1 + x_2 < R$, then the $R - x_1 - x_2$ bitcoins from the block reward become unspendable. So coinbase transactions set the sum of the output amounts to be equal to the block reward. In the past, errors in coinbase transaction creation by miners have resulted in blocks where this sum is not equal to the block reward.

To spend the bitcoins earned in a coinbase transaction, the miner would have to create a regular transaction. Regular transactions have at least one input and at least one output. Each input specifies three items:

- The transaction identifier (TXID) of a previous transaction on the blockchain. The TXID of a transaction is its double SHA-256 hash.
- The index of an output in the previous transaction. The first output in a transaction has index 0, the second output has index 1 and so on.
- A response script which will satisfy the conditions required to spend the bitcoins in the output.

Essentially each input in a regular transaction unlocks the bitcoins associated with a previous transaction output. This previous transaction could be a coinbase transaction or a regular transaction. Note that the inputs do not specify the amount of bitcoins to be spent from an output. If an output of a previous transaction is referenced by an input, all the bitcoins associated with that output need to be spent in the transaction.

The outputs in a regular transaction have the same format as the outputs in a coinbase transaction. Each of them specifies an amount of bitcoins being associated with that output and a challenge script. The amounts in the regular transaction outputs can take any value as long as the sum of the amounts does not exceed the total amount of bitcoins unlocked by the inputs. Suppose a regular transaction has N inputs and M outputs. Let the ith input unlock x_i bitcoins from a previous transaction output. Then $\sum_{i=1}^{N} x_i$ bitcoins will be available for transfer from all the inputs. Let the jth output specify an amount of y_j bitcoins. The transaction is valid if $\sum_{j=1}^{M} y_j \leq \sum_{i=1}^{N} x_i$. The constraint ensures that the amount of outgoing bitcoins is at most the amount of incoming bitcoins. *But why are these amounts not equal?* The difference in the amounts is the transaction fees paid to the miner which includes this transaction in a block, i.e.

$$\text{Transaction fees} = \sum_{i=1}^{N} x_i - \sum_{j=1}^{M} y_j.$$

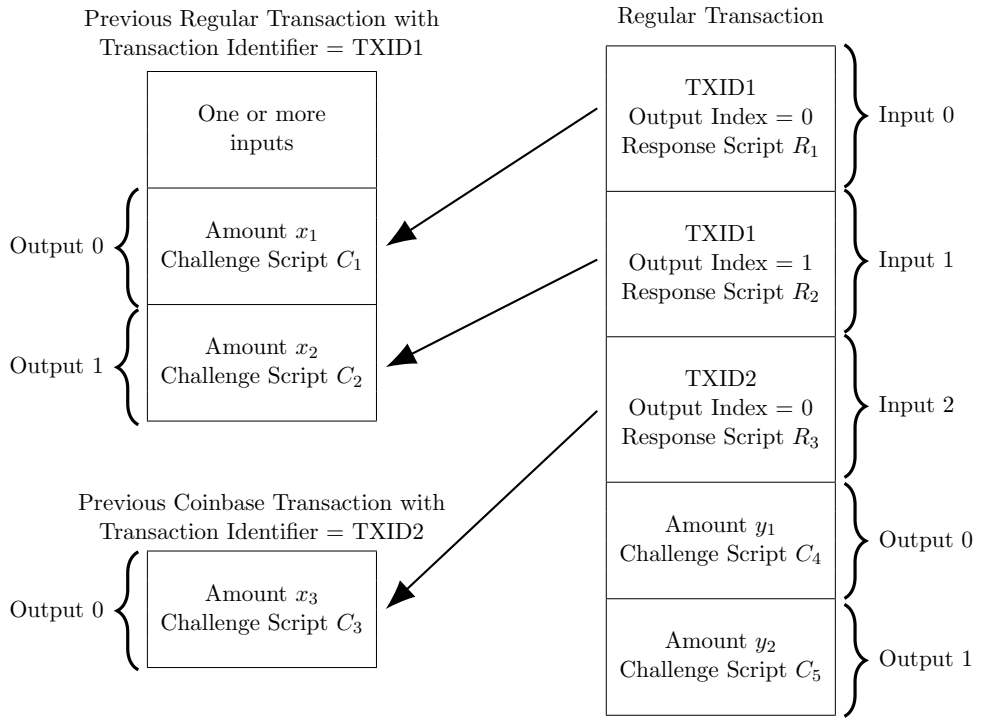


Figure 4.10: Illustration of a regular transaction spending outputs from two previous transactions.

Figure 4.10 illustrates a regular transaction which has three inputs and two outputs. The first two inputs refer to two outputs from a previous regular transaction with transaction identifier equal to TXID1. The third input refers to the output of a previous coinbase transaction with transaction identifier equal to TXID2. The first output in the previous regular transaction specifies a bitcoin amount x_1 and a challenge script C_1. The first input of the regular transaction unlocks the x_1 bitcoins by providing a satisfactory response R_1 to C_1. Similarly, the second and third inputs of the regular transaction unlock x_2 and x_3 bitcoins by providing satisfactory responses R_2 and R_3 to challenges C_2 and C_3 respectively. A total of $x_1 + x_2 + x_3$ bitcoins are available for transfer to the outputs. The two outputs of the regular transaction are allocated y_1 and y_2 bitcoins respectively, where $y_1 + y_2 \leq x_1 + x_2 + x_3$. Challenge scripts C_4 and C_5 are included in the outputs specifying conditions under which these outputs can be spent. A transaction fee of $x_1+x_2+x_3-y_1-y_2$ can be claimed by the miner which includes this transaction in a block on the blockchain.

How is the value of the transaction fee determined? Miners aim to maximize their block reward while constructing candidate blocks. As the block subsidy is fixed, they seek to maximize the sum of the transaction fees from the transactions included in the block. A high transaction fee is not the only

factor in determining which transaction to include in a block. The size of the transaction in bytes also matters as the block size is constrained to be at most one megabyte. A transaction t_1 may pay a lower transaction fee than a transaction t_2 but t_1 may have a smaller size in bytes allowing for more transactions to be included in the block. Miners divide the transaction fee offered by a transaction by the transaction size in bytes to get the *transaction fee per byte* or the *fee rate* of the transaction. While collecting regular transactions to form a candidate block, they give higher priority to transactions which pay a higher fee rate. Nodes which want to add their regular transactions to the blockchain need to pay a fee rate which is competitive with the other transactions being broadcast on the network. The probability that a transaction paying a certain fee rate r will be included in the next m blocks can be estimated by observing the fee rates paid by the transactions being included in the blockchain. When a transaction t with fee rate r is first heard by a node, the height h_1 of the latest block in the blockchain is recorded. If t gets included in a block on the blockchain at height h_2, the difference between h_2 and h_1 gives an estimate of the expected delay incurred by a transaction which pays fee rate r. Using such estimates from several transactions, a competitive fee rate can be estimated as a function of the amount of delay in blocks the node creating the transaction is willing to tolerate. The Bitcoin Core software implements such a fee estimation algorithm.

4.5 Bitcoin Ownership

When an output of a previous transaction is unlocked by the input of a later transaction, all the bitcoins in the output need to be spent. This implies that a transaction output can be in only one of two states: spent or unspent. *Unspent transaction outputs (UTXOs)* refer to outputs in transactions recorded on the blockchain which have not been unlocked by the inputs of later transactions. When a new block is added to the blockchain, the output of the coinbase transaction is a UTXO containing the block reward. Every regular transaction in the new block unlocks UTXOs from transactions in previous blocks and creates new UTXOs. The unlocked outputs cease to be UTXOs. So the set of UTXOs changes with every block addition.

The set of all UTXOs on the blockchain represent an ownership record of all the bitcoins in circulation. Suppose a UTXO has x bitcoins and a challenge script C in it. An entity which can unlock the UTXO by providing a satisfactory response R to C is considered the owner of the x bitcoins. *The total amount of bitcoins owned by an entity is the sum of the amounts in all the UTXOs it can unlock.*

During a blockchain fork, different network nodes may consider different branches in the fork to be the longest branch. Since a node calculates the set of UTXOs from its local copy of the blockchain, the UTXO set will differ

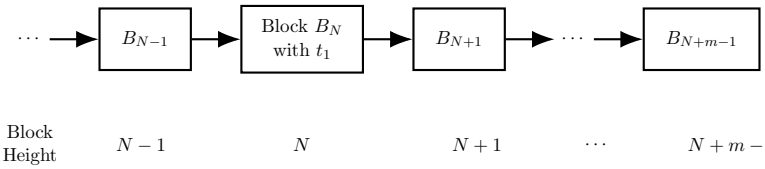
across nodes whose local copies differ. For example, consider the blockchain fork illustrated in Figure 4.8(a) where miners M_A and M_B have mined blocks A and B respectively. The UTXO set in nodes which consider the branch containing block A to be the longest branch will contain the output of the coinbase transaction which transfers the block reward to M_A. This output will not be present in nodes which consider the branch ending in block B to be the longest branch. The UTXO set in these nodes will contain the output of the coinbase transaction which transfers the block reward to M_B. Hence the ownership of the block reward in the block at height N is undetermined until the fork is resolved. Once blockchain forks are resolved, the UTXO set (and consequently the bitcoin ownership record) seen by all the nodes in the network will be identical. But the temporary ambiguity about the UTXO set during blockchain forks should be taken in account in situations where bitcoins are used as a mode of payment.

4.6 Double Spending Attacks

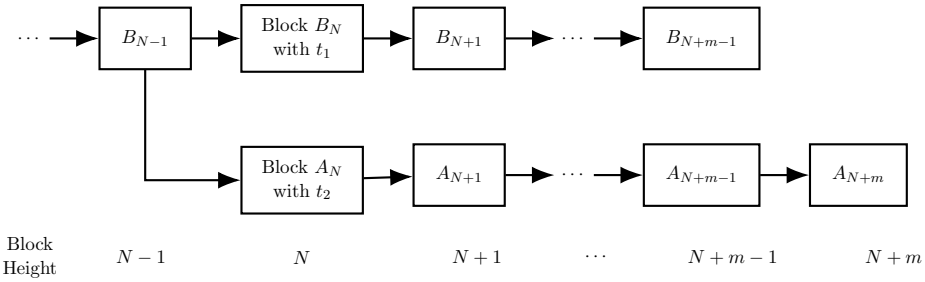
Suppose Alice wants to use bitcoins to pay for some goods which Bob is selling. In order to pay Bob, Alice needs to unlock some UTXOs she controls and create a new UTXO which only Bob can unlock. For Alice to be able to create the new UTXO, Bob needs to provide Alice with a Bitcoin address he owns. Suppose Bob provides Alice with a P2PKH address for which he knows the corresponding private key. Alice will then create a new regular transaction t_1 where one of the outputs contains the amount of bitcoins Alice wants to pay Bob. The challenge script in this output will contain the P2PKH address. Constructing a satisfactory response to this challenge will require the private key known only to Bob. Alice will broadcast t_1 on the Bitcoin network where miners will include it in the candidate blocks they are mining. Bob will keep scanning the new blocks being added to the blockchain for t_1. Once Bob sees a new block with t_1 included in it, he will wait for the branch containing this block to grow further before handing the over the goods to Alice. This is illustrated in Figure 4.11(a). Suppose the transaction t_1 is included in a block B_N at height N. The t_1 is said to have received one *confirmation*. When a valid block B_{N+1} at height $N+1$ is added to the blockchain with B_N as its previous block, t_1 is said to have received two confirmations. Before transferring the goods to Alice, Bob waits until t_1 receives m confirmations, i.e. the branch containing t_1 has been extended by $m-1$ blocks. This is to safeguard against a *double spending attack* by Alice in which the transaction t_1 can be cancelled.

Suppose Bob asks Alice for x bitcoins as payment for the goods. Also suppose that Alice can unlock a UTXO O_A which has at least x bitcoins. A double spending attack by Alice proceeds as follows.

1. Alice creates two transactions t_1 and t_2. In t_1, an input unlocks O_A and an



(a) Block chain state when Bob hands over the goods to Alice (after t_1 receives m confirmations)



(b) Block chain state after Alice succeeds in creating a longer branch containing t_2

Figure 4.11: Illustration of a double spending attack

output pays Bob x bitcoins. In t_2, an input unlocks O_A but the x bitcoins are paid back to Alice in an output. The transactions t_1 and t_2 conflict because they both spend the same UTXO O_A. Only one of them can be included in the blockchain.

2. Alice will broadcast t_1 on the Bitcoin network for inclusion in the blockchain. She will keep t_2 a secret.

3. Suppose a miner includes t_1 in a valid block B_N which is at height N. Bob will wait for t_1 to receive m confirmations before transferring the goods to Alice.

4. Immediately after broadcasting t_1, Alice begins work on constructing a branch containing t_2. She does not announce the valid blocks found on this branch to the Bitcoin network before Bob transfers the goods to her.

5. After receiving the goods, if Alice succeeds in creating a branch containing t_2 which is longer than the branch containing t_1 then she broadcasts all the blocks in the t_2 branch in the Bitcoin network. This is illustrated in Figure 4.11(b) where the branch containing t_2 is one block longer than the branch containing t_1. The blocks $A_N, A_{N+1}, \ldots, A_{N+m}$ are broadcasted by Alice after she receives the goods from Bob.

6. All the nodes in the Bitcoin network will eventually switch to the t_2 branch and the t_1 branch will be abandoned. Usually, transactions which are in

stale blocks, i.e. blocks which are in abandoned branches, are added back to the transaction pool if they have not already appeared in the surviving branch. Miners use this transaction pool for constructing new candidate blocks. However, miners which have switched to the t_2 branch will not add t_1 to their transaction pools as it conflicts with t_2. The end result is that Bob has already transferred the goods to Alice but the x bitcoins he thought he received from Alice in t_1 are back in Alice's possession. Since Alice can now spend these bitcoins again, this attack is called a double spending attack.

The double spending attack as described above will always succeed if Alice can influence 50% or more of the total network hash rate to work on the branch containing t_2. With the majority of the network hash rate working to extend it, the t_2 branch will eventually overtake the t_1 branch. Alice could herself be a miner who controls a majority of the network hash rate or she could collude with a group of miners who collectively control a majority of the network hash rate. If less than 50% of the network hash rate is used to attempt a double spending attack, it may or may not succeed. Suppose that a fraction q of the network hash rate is used to mount the double spending attack where $q < \frac{1}{2}$ and that the remaining $p = 1 - q$ fraction of the network hash rate continues to extend the branch containing t_1. If Bob waits for m confirmations before transferring the goods to Alice, then the success probability of the double spending attack is

$$P(m, q) = 1 - \sum_{k=0}^{m} \binom{m+k-1}{k} \left[p^m q^k - p^{k-1} q^{m+1} \right]. \tag{4.1}$$

It is derived in Appendix B. Table 4.2 lists $P(m, q)$ for some values of m and q. For all values of q less than 0.5, $P(m, q)$ decreases with increasing m. So waiting for more confirmations reduces the risk of a successful double spending attack. While the rate of decrease in $P(m, q)$ is exponential for $q = 0.1$, it slows down as q increases. For $q = 0.49$, a double spending attack has a 95% chance of success if Bob waits for only one confirmation. Even if Bob waits for ten confirmations, a double spending attack has a 90% chance of success. For a fixed value of m, $P(m, q)$ increases exponentially with q, eventually becoming one when q exceeds 0.5.

In spite of the guaranteed success of a double spending attack mounted using a majority of the network hash rate, such attacks have not been observed in practice. This is probably because successful double spending attacks would undermine confidence in the Bitcoin currency as a mode of payment and reduce its exchange rate in terms of fiat currency. The miners who control significant fractions of the network hash rate have invested large amounts of capital in establishing their mining infrastructure. These investments will continue to generate a profit for the miners when the bitcoins earned as part of the block

m	$q = 0.1$	$q = 0.2$	$q = 0.3$	$q = 0.4$	$q = 0.49$	$q = 0.5$
1	0.031111	0.130000	0.308571	0.586667	0.950984	1
2	0.009511	0.072400	0.232971	0.529067	0.943637	1
3	0.003031	0.041680	0.180051	0.482987	0.937517	1
4	0.000990	0.024477	0.141155	0.444279	0.932164	1
5	0.000329	0.014568	0.111750	0.410836	0.927348	1
6	0.000110	0.008754	0.089107	0.381407	0.922936	1
7	0.000037	0.005300	0.071446	0.355172	0.918840	1
8	0.000013	0.003227	0.057538	0.331561	0.915002	1
9	0.000004	0.001974	0.046505	0.310153	0.911378	1
10	0.000002	0.001212	0.037700	0.290630	0.907937	1

Table 4.2: Success probability of a double spending attack as a function of m and q.

reward can be sold at a high price. So these miners will avoid any behaviour, like attempting double spending attacks, which may negatively affect the price. Nevertheless, double spending attacks are still possible by an attacker who does not have a long term stake in Bitcoin. For example, a hacker may be able to temporarily take control of a miner node and divert its hash rate toward mounting a double spending attack. So it is prudent for merchants like Bob to wait for confirmations on a transaction before considering it valid.

How many confirmations m should Bob wait for? Bob does not know the fraction of network hash rate q which will be used by Alice to mount a double spending attack. If $q \geq \frac{1}{2}$, then the attack will be successful irrespective of the value of m. By accepting bitcoins as a valid mode of payment, Bob is implicitly assuming that Alice cannot gain control over a majority of the network hash rate. If $q < \frac{1}{2}$, then the success probability of the double spending attack decreases as m increases. As he does not know the value of q, Bob cannot choose the value of m to bring the success probability below a predetermined level like 0.01. He can only hope to reduce the success probability by increasing m. But m cannot be very large as each confirmation takes approximately 10 minutes to appear. Consequently, all customers, irrespective of whether they are honest or malicious, will experience a delay of about $10m$ minutes before Bob transfers the goods to them. Several merchants in the Bitcoin ecosystem wait for six confirmations ($m = 6$), which corresponds to a delay of about an hour before goods are transferred from a merchant to a customer. Smaller or larger values of m are used by merchants depending on the value of the goods being sold.

A *zero confirmation transaction* is one which has been broadcast on the Bitcoin network but has not been included in a valid block on the blockchain. When the value of goods involved is small and confirmation delays cannot be tolerated, merchants may accept zero confirmation transactions as valid

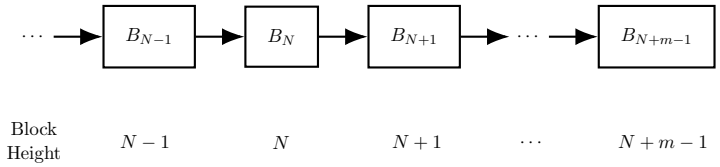
payment. For example, suppose Bob runs a coffee shop where bitcoins can be used to buy coffee. To pay for a cup of coffee, Alice broadcasts a transaction on the Bitcoin network paying Bob the required amount of bitcoins. Bob may choose to give Alice her coffee as soon as he hears the transaction on the network and before it has been included in a valid block. Since this transaction has zero confirmations, it can be cancelled more easily through a double spending attack. But Bob may take this risk because it is unlikely that Alice will undertake the effort involved in a double spending attack just for a cup of coffee's worth of bitcoins. Also, the delay incurred in waiting for even one confirmation may be undesirable in case Alice is an honest customer.

4.7 Blockchain Integrity

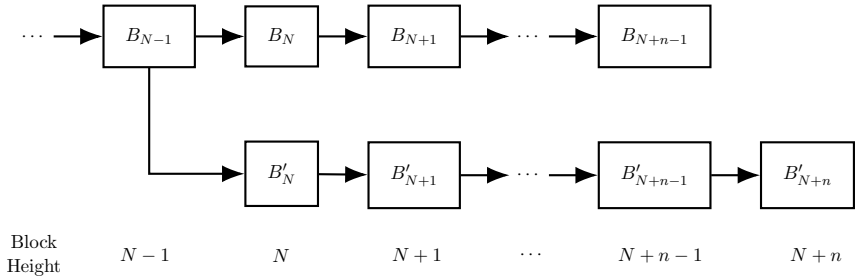
Suppose Alice wants to modify an existing block B_N which is at height N in the blockchain and has received m confirmations.[4] Let $B_{N+1}, B_{N+2}, \ldots, B_{N+m-1}$ be the blocks which succeed B_N on the blockchain as shown in Figure 4.12(a). Alice may want to delete a transaction from B_N, add a transaction to B_N, or just modify the timestamp in B_N's block header. Let B'_N be the block after the modifications have been made to B_N. Modifying the timestamp clearly causes the block headers of B'_N and B_N to be different. Adding or deleting a transaction causes the `hashMerkleRoot` fields in the block headers of B'_N and B_N to be different. If the block hash of B'_N does not below the target threshold, Alice will have to perform mining on B'_N until it finds a `nNonce` value which makes B'_N a valid block. To replace B_N with B'_N in all the copies the blockchain stored across the Bitcoin network, Alice has to create a branch containing B'_N which is longer than the branch containing B_N (as illustrated in Figure 4.12(b)) and broadcast it. Once all the nodes in the Bitcoin network switch to the branch containing B'_N, it will become the block at height N in the blockchain. This strategy of constructing a longer branch is also used in double spending attacks. The difference is that here the B_N branch has a lead of m blocks when Alice begins mining the B'_N branch. In the double spending attack scenario, the t_1 branch has no lead in terms of blocks when Alice begins mining the t_2 branch.

Even though Alice is interested in only modifying the block at height N, she has to construct new valid blocks at heights $N + 1$, $N + 2$ and so on. This is because the block B'_N is not a drop-in replacement for B_N in the blockchain. The block header of B_{N+1} contains the block hash of B_N in the `hashPrevBlock` field. As the block headers of B_N and B'_N differ, their block hashes also differ by the collision resistance of the SHA-256 hash function. So B_{N+1} is not a valid successor block to B'_N. Alice has to mine a new valid block B'_{N+1} which has the block hash of B'_N in its `hashPrevBlock` field. By

[4] A block is said to have received m confirmations if the transactions in it have received m confirmations as defined in Section 4.6.



(a) Block chain state before Alice attempts to modify B_N



(b) Block chain state when the branch containing B'_N overtakes the branch containing B_N. Here $n \geq m$.

Figure 4.12: Illustration of the work required to modify an existing block

the same argument, B_{N+2} is not a valid successor to B'_{N+1} and Alice has to mine a new valid block B'_{N+2} which has the block hash of B'_{N+1} in its `hashPrevBlock` field.

While Alice is mining blocks on the B'_N branch, the other miners will continue mining blocks on the B_N branch as it is the longest branch they know. Let q be the fraction of the total network hash rate which Alice controls. If $q \geq \frac{1}{2}$, then Alice will eventually succeed in constructing the longer branch irrespective of the value of m. If $q < \frac{1}{2}$, then the probability[5] that Alice succeeds is $(\frac{q}{1-q})^{m+1}$ where m is the number of confirmations B_N has received when Alice begins mining the B'_N branch. As this probability decreases exponentially with m, blocks become more difficult to tamper with as the number of confirmations they have received increases. Table 4.3 lists Alice's success probability for some values of m and q. Unless Alice controls a fraction of the network hash rate which is close to 0.5, it becomes nearly impossible for her to tamper with blocks which have received 50 or more confirmations. Even with $q = 0.4$, Alice only has a one in a billion chance of modifying a block with 50 confirmations. Given that a block receives about six confirmations in an hour, a block with 50 confirmations is about eight hours old.

Even if Alice controls the majority of the network hash rate, she cannot make modifications to existing blocks which require her to know the private keys of other users. For example, suppose a block contains a transaction where Bob transfers some bitcoins to Carol. Such a transaction will have an input which unlocks a UTXO owned by Bob and an output which creates a UTXO

[5]See Appendix B for the derivation.

m	$q = 0.1$	$q = 0.2$	$q = 0.3$	$q = 0.4$	$q = 0.49$
1	0.012346	0.062500	0.183673	0.444444	0.923106
5	1.88×10^{-6}	0.000244	0.006196	0.087791	0.786603
10	3.19×10^{-11}	2.38×10^{-7}	8.96×10^{-5}	0.011561	0.643999
20	9.14×10^{-21}	2.27×10^{-13}	1.87×10^{-8}	0.000200	0.431662
50	2.16×10^{-49}	1.97×10^{-31}	1.71×10^{-19}	1.05×10^{-9}	0.129993
100	4.18×10^{-97}	1.56×10^{-61}	6.83×10^{-38}	1.64×10^{-18}	0.017588

Table 4.3: Success probability of block modification as a function of m and q.

that can be unlocked only by Carol. Alice cannot modify this transaction to make herself the recipient of the bitcoins instead of Carol. This is because the response script Bob uses to unlock his UTXO requires a digital signature which can only be generated using Bob's private key. The output of the transaction which specifies Carol as the recipient is part of the the message that is used to generate the signature. If Alice replaces this output with an output which specifies her as the recipient, the message used to generate the signature changes and Bob's private key is needed to generate the new signature. Furthermore, Alice cannot even change the amount of bitcoins which Bob is transferring to Carol as this amount is also part of the message that is used to generate the signature. A more detailed description of the signature generation procedure is given in Section 5.6.

4.8 The 51% Attacker

An attacker who controls 50% or more of the network hash rate is called a *51% attacker* in the Bitcoin literature. This is a slight misnomer as the actual percentage of the network hash rate controlled by the attacker may differ from 51%. As explained in the last two sections, a 51% attacker can successfully mount double spending attacks and modify blocks in the blockchain irrespective of the number of confirmations. But these two attacks are minor compared to the other attacks a 51% attacker can mount.

Consider the situation where a 51% attacker performs mining like a regular miner node with the exception that she does not switch to longer branches which are announced by the rest of the network. As the branch mined by the attacker will eventually become the longest branch of the blockchain, all new bitcoins generated as part of the block subsidy will be owned by her. This will make mining financially unviable for the other miners in the network. If these miners stop mining, the attacker may end up becoming the only miner in the network. The Bitcoin system will then resemble a centralized system controlled by the 51% attacker. The attacker can unilaterally decide which transactions get recorded on the blockchain. For instance, the attacker can censor transactions which transfer bitcoins to a merchant by not including

such transactions in new blocks. She can also decide the minimum fee rate for transactions by not including those transactions which pay less than this minimum into new blocks. Such behaviour will cause the Bitcoin system to become less attractive as a mode of payment. The attacker can even cause the Bitcoin system to stop functioning as a payment system by mining only *empty blocks*. These are blocks which contain only the coinbase transaction and no regular transactions. Such blocks are considered valid by the Bitcoin protocol. Without any new regular transactions appearing on the blockchain, the Bitcoin currency would be worthless.

While the presence of a 51% attacker can lead to the collapse of the Bitcoin system, such an event is unlikely due to the prohibitive costs involved in generating a majority of the network hash rate. Even if the cost of acquiring the required mining equipment can be ignored, the electricity and cooling costs involved in keeping the equipment running will be high. The 51% attacker cannot hope to recover these costs by selling bitcoins as the attack itself will drive the Bitcoin price down to zero by undermining the effectiveness of the Bitcoin system.

4.9 Summary

The blockchain is the main innovation in Bitcoin. By incentivizing addition of blocks to the blockchain, the Bitcoin system ensures that multiple copies of the blockchain are maintained across a geographically distributed network. The network achieves consensus over the state of the blockchain by having each node switch to the longest branch it hears. The computationally demanding task of mining valid blocks not only ensures a predictable rate of new currency creation but also makes the whole system resistant to control by a single entity. With attackers not able to control significant fractions of the network hash rate, double spending attacks become unlikely and transactions with a few dozen confirmations can be considered irreversible.

Chapter 5

Bitcoin Transactions

In this chapter, we describe the format of Bitcoin transactions including the format of the challenge and response scripts. Our description assumes that the reader is familiar with the high-level description of transactions given in Section 4.4. The original transaction format introduced by Satoshi Nakamoto in 2009 was the only valid transaction format until August 2017, when a set of changes to the Bitcoin protocol called Segregated Witness (SegWit) was activated. SegWit added a new transaction format to solve the problem of *transaction malleability* which affected the original transaction format. For clarity and convenience, we will call the original transaction format the *pre-SegWit transaction format*. This transaction format continues to be valid after SegWit activation.

We will present the pre-SegWit transaction format first and discuss the structure of the challenge and response scripts supported by it. This will enable us to illustrate the problem of transaction malleability, whose solution was one of the main motivations for SegWit. It will also help us motivate the design of the SegWit transaction format, which will appear unnatural without an understanding of the pre-SegWit transaction format. SegWit is a *soft fork* change[1] to the Bitcoin protocol that requires SegWit transactions to look like valid pre-SegWit transactions to network nodes which are not running SegWit-capable client software. The SegWit transaction format has been cleverly designed to accommodate this constraint.

5.1 Block Format

Before delving into the pre-Segwit transaction format, let us discuss the block format. Each block in the blockchain begins with a 80-byte block header (see Section 4.2). The field immediately following the block header encodes the number of transactions n included in the block as shown in Figure 5.1. This is

[1]For details on how new features are added to the Bitcoin protocol via soft forks, see Chapter 7.

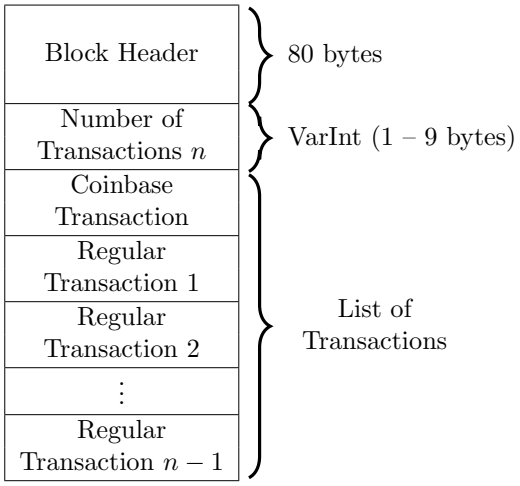


Figure 5.1: Block format

followed by a list of transactions which must begin with a coinbase transaction. The coinbase transaction is mandatory in a block as it encodes the transfer of the block subsidy to the miner who mined the block. Regular transactions are optional as it is possible that there are no regular transactions available to be included in a block. While such a scenario is unlikely today due to the popularity of Bitcoin, it was common in the early days of Bitcoin. Each of the regular transactions in a block can be either a pre-SegWit or a SegWit transaction. If all the regular transactions are pre-SegWit transactions then the coinbase transaction also adheres to the pre-SegWit transaction format. Otherwise, the coinbase transaction adheres to the SegWit transaction format.

The number n is encoded by a variable length integer (VarInt) which occupies 1, 3, 5, or 9 bytes depending on its value. With fixed length encoding, one needs 8 bytes to represent an integer in the range 0 to $2^{64}-1$. The VarInt encoding uses 5 bytes or less for the integers in this range 0 to $2^{32}-1$, as shown in Table 5.1. It is a more efficient encoding if smaller integers are more likely. The VarInt encoding proceeds as follows:

- If $n \in \{0, 1, \ldots, 252\}$, encode n as a 8-bit unsigned integer.
- If $n \in \{253, 254, \ldots, 2^{16}-1\}$, encode n using three bytes. The first byte contains the prefix 253 followed by the representation of n as a 16-bit unsigned integer.
- If $n \in \{2^{16}, 2^{16}+1, \ldots, 2^{32}-1\}$, encode n using five bytes. The first byte contains the prefix 254 followed by the representation of n as a 32-bit unsigned integer.

Integer Range	VarInt Encoding Prefix	VarInt Size (bytes)
0 to 252	None	1
253 to $2^{16} - 1$	253	3
2^{16} to $2^{32} - 1$	254	5
2^{32} to $2^{64} - 1$	255	9

Table 5.1: Representation of integers from 0 to $2^{64} - 1$ using a VarInt

- If $n \in \{2^{32}, 2^{32} + 1, \ldots, 2^{64} - 1\}$, encode n using nine bytes. The first byte contains the prefix 255 followed by the representation of n as a 64-bit unsigned integer.

Decoding an integer stored in VarInt form is trivial as the first byte tells us the length of the encoding. The VarInt encoding is used frequently in the Bitcoin transaction format to specify the lengths of lists or variable length fields.

5.2 Pre-SegWit Regular Transactions

The format of a pre-SegWit regular transaction is shown in Figure 5.2. The field names in teletype font (like `nVersion`) are from the Bitcoin Core reference client.

Transaction Version

The transaction begins with a 4-byte field called `nVersion` which is used to store the version number of the transaction format. The version number dictates the rules for interpreting the fields in a transaction. As of August 2017, the transaction version number can be either 1 or 2. The two versions differ in the interpretation of the `nSequence` field in the transaction inputs (to be discussed later).

Input and Output Lists

The second field in the transaction is a VarInt encoding of the number of inputs N. This field is followed by the N inputs. The list of inputs is followed by the number of outputs M stored as a VarInt, which in turn is followed by the M outputs.

Transaction Lock Time

The final field in the transaction is a 4-byte field called `nLockTime`. It stores a *lock time* for the transaction which is the earliest time that the transaction

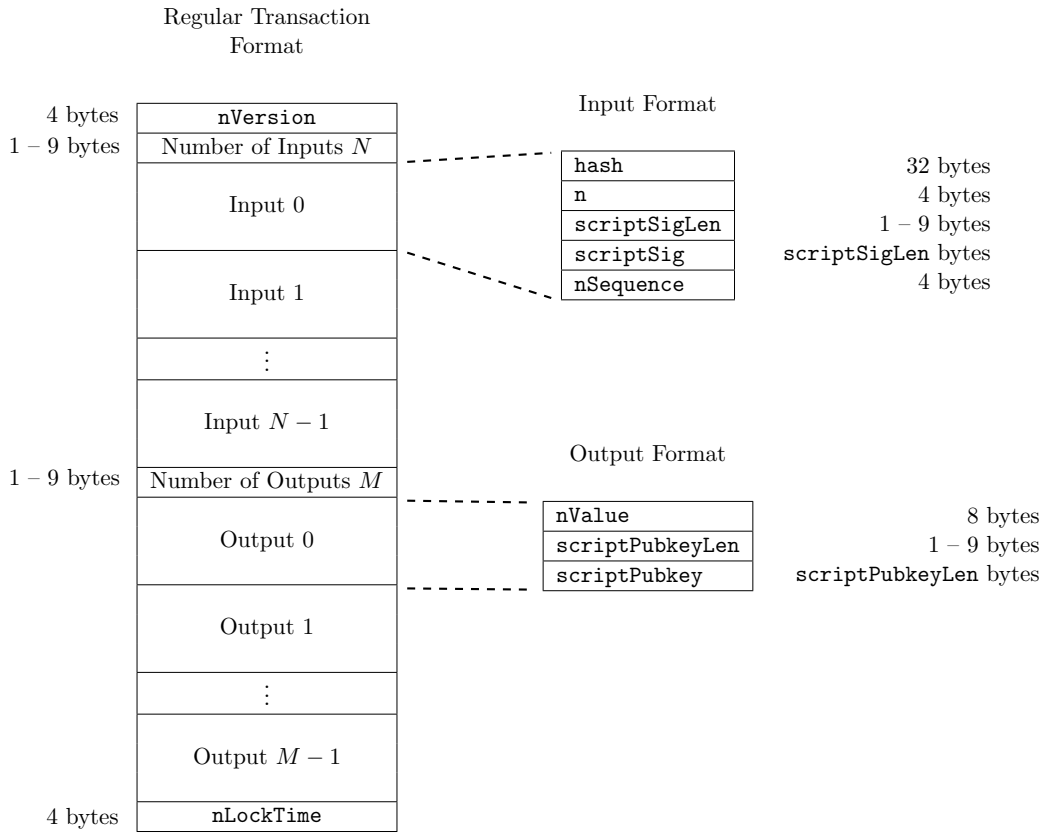


Figure 5.2: Pre-SegWit regular transaction format

can be included in a block on the blockchain. The motivation behind this design was to enable the creation of smart contracts (see Chapter 6). The lock time can be specified either in terms of block height or Unix time.[2]

- If `nLockTime` $< 5 \times 10^8$, then it is interpreted as a block height. For example, a transaction with `nLockTime` = 600,000 will not be included in a valid block whose height is less than 600,000. When such a transaction is broadcast on the network, it will be rejected by the network nodes unless the height of the next block to be added to the blockchain has height 600,000 or more.
- If `nLockTime` $\geq 5 \times 10^8$, then it is interpreted as a Unix time. For example, the value `nLockTime` = 1,514,797,200 corresponds to 9:00 AM on January 1, 2018. A transaction having an `nLockTime` field interpreted as Unix time will not be considered for inclusion in the next block on the blockchain unless the median-time-past of the latest block in the

[2]See discussion about the `nTime` field in Section 4.3 for details about the Unix time.

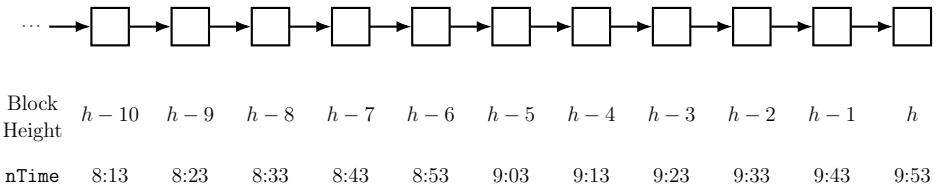


Figure 5.3: Illustration of the one hour delay in the actual lock time due to usage of median-time-past to determine expiry.

blockchain exceeds the `nLockTime` value. Recall that the median-time-past of a block at height h is the median of the `nTime` values in the 11 blocks at heights $h, h-1, \ldots, h-10$.

What if we want to have a lock time at block height which is greater than 5×10^8 *or at a Unix time which is less than* 5×10^8*?* Given that 2,016 blocks are mined approximately every two weeks, it would take more than 9,500 years for the block height to exceed 5×10^8. Not being able to specify lock times that far ahead in the future is effectively not a restriction. The Unix time of 5×10^8 corresponds to a time in the past (12:53 AM on November 5, 1985). Thus a lock time that expires at a Unix time which is less than 5×10^8 will never be required.

The policy of using the median-time-past to check `nLockTime` validity was proposed in BIP 113. It was made mandatory from the block at height 419328 which was mined in July 2016. Prior to that, a transaction was allowed to be part of a block if the `nTime` value in the block's header exceeded the `nLockTime` value. This was changed because miners could set the `nTime` in a candidate block to a future value in order to include transactions whose lock times had not yet expired and increase the transaction fees they received. As the median of the `nTime` values is known to all the miners, manipulating it by setting the `nTime` value to a future value in mined blocks does not give any specific miner an advantage over the other miners. Another advantage of comparing `nLockTime` to median-time-past values instead of `nTime` values is that median-time-past values increase monotonically with block height while `nTime` values are not required to. So if the lock time of a transaction expires at a certain block height, it remains in the expired state irrespective of the `nTime` values in the blocks at subsequent heights.

A consequence of using the median-time-past is that the lock time of a transaction expires approximately one hour after the time specified in the `nLockTime` field. For example, suppose the `nLockTime` field of a transaction t specifies a Unix time corresponding to 9:00 AM on January 1, 2018. Figure 5.3 illustrates the state of the blockchain when the lock time of this transaction expires. The latest block in the blockchain has height h and an `nTime` value of 9:53 AM on January 1, 2018. The `nTime` values in the 11 blocks at heights $h - 10$ through h have been chosen to be exactly 10 minutes apart for the

sake of illustration (the AM suffix and date have been omitted in the figure). The median-time-past of the block at height h is 9:03 AM on January 1, 2018 corresponding to the `nTime` of the block at height $h - 5$. So the transaction t can be included in the next block at height $h + 1$. Note that the lock time of the transaction t expired after the block at height h was added to the blockchain. Since the `nTime` field is populated before the mining of the candidate block begins, the block at height h was probably broadcast on the network approximately 10 minutes after 9:53 AM. So the effective time of expiry of the lock time on the transaction t is around 10:00 AM rather than 9:00 AM on January 1, 2018.

For the `nLockTime` to be considered while evaluating a transaction for inclusion in a block, at least one of the transaction inputs must have an `nSequence` value which is less than `0xFFFFFFFF`. The `nSequence` field occupies 4 bytes and can have a maximum value of `0xFFFFFFFF`. If all the inputs have `nSequence` value equal to `0xFFFFFFFF`, then the `nLockTime` field is ignored and the transaction can be included in any block.

If the lock time applies to the whole transaction, why is its validity controlled by the `nSequence` fields in all the transaction inputs? Why not use a single field to indicate whether the `nLockTime` should be ignored or not? The `nSequence` field was originally intended to enable multiple entities to collaboratively construct a multi-input transaction where each entity was responsible for one of the inputs. The initial version of the transaction would have a lock time in the future and `nSequence` values less than `0xFFFFFFFF`. The entities would indicate a newer version of a transaction by increasing the `nSequence` values in the inputs owned by them. The original Bitcoin Core reference client implementation allowed the replacement of a transaction in a node's transaction mempool with newer versions until the transaction's lock time expired. A transaction was considered final once all inputs had `nSequence` values equal to `0xFFFFFFFF` and could be included in a block immediately without waiting for the lock time to expire. Transaction replacement based solely on `nSequence` values was disabled in 2010 as a malicious node could flood the network with newer versions of the same transaction without incurring any penalty. There was also no way to guarantee that miners would include a newer version of a transaction in a block when the older version paid a higher transaction fee. When disabling transaction replacement, the transaction format was not changed and the `nSequence` field in each input continued to control the validity of the transaction lock time.[3]

[3] A fee-based transaction replacement policy called *opt-in full replace-by-fee (RBF)* which does not suffer from the shortcomings of the `nSequence`-based transaction replacement was proposed in BIP 125. An implementation of opt-in full RBF was added to the Bitcoin Core reference client in 2016. But it is a policy which does not affect block validity and is not required to be strictly followed by the nodes in the network.

Input Format

Each input in a pre-SegWit regular transaction has the same five fields. Figure 5.2 shows these fields for the first input. The fields in the other inputs are not shown for brevity. The input fields have the following semantics.

- The `hash` field contains the 256-bit transaction identifier (TXID) of a previous transaction containing the output which will be unlocked by this input. The field is called `hash` because the TXID is the double SHA-256 hash of the previous transaction.
- The 4-byte `n` field contains the index of the output being unlocked in the previous transaction. The index of the first output in the previous transaction is 0, the index of the second output is 1, and so on.
- The `scriptSigLen` field is a VarInt encoding of the length of the response script which is used to unlock the output.
- The `scriptSig` field contains the response script itself. The format of response and challenge scripts will be described in Section 5.4.
- The 4-byte `nSequence` field is interpreted differently depending on the transaction version. In both version 1 and version 2 transactions, if all transaction inputs have their `nSequence` fields set to `0xFFFFFFFF`, then the `nLockTime` field is ignored.

  In version 2 transactions, the `nSequence` field can be used to specify a *relative lock time* of an input. The relative lock time can have units of either number of blocks or seconds. Before we go into the details of the encoding, let us consider the functionality of the relative lock time. Suppose the relative lock time of an input is k blocks. If the output which is being unlocked by this input is in a block with height K, then a transaction containing this input cannot be included in a block whose height is less than $K+k$. Now suppose the relative lock time of the input is specified as t seconds. If the output being unlocked by this input is in a block at height h_o, let the median-time-past of the block at height $h_o - 1$ be T seconds. Then a transaction containing the input cannot be included in a block at height h_i until the median-time-past of the block at height $h_i - 1$ exceeds $T + t$ seconds.

  The relative lock time is unlike the lock time specified by `nLockTime` which specifies an absolute block height or median-time-past before which a transaction cannot be included in a block. The relative lock time was introduced in BIP 68 in order to enable smart contracts which involve a sequence of dependent transactions with minimum delays between them (see Chapter 6). Absolute lock times cannot be used to ensure a delay between two transactions if the time or block height at which the first transaction is added to the blockchain is not known.

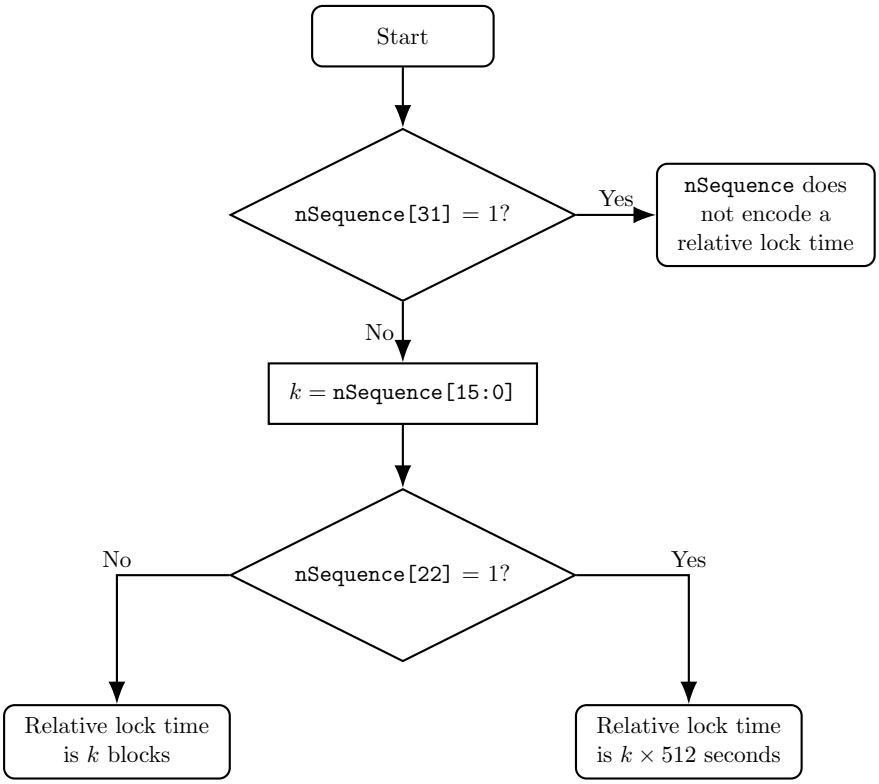


Figure 5.4: Relative lock time encoding in the `nSequence` field of version 2 transactions

The flowchart in Figure 5.4 illustrates the relative lock time encoding in version 2 transactions. Let `nSequence[i]` denote the bit with index `i` in the field where `nSequence[0]` is the least significant bit (LSB) and `nSequence[31]` is the most significant bit (MSB). If `nSequence[31]` is set, then the `nSequence` field does not encode a relative lock time. If it is not set, then the `nSequence` field encodes a relative block time whose units are determined by the bit `nSequence[22]`. The magnitude of the relative lock time is given by the 16 least significant bits `nSequence[15]` to `nSequence[0]`. Let k be the unsigned integer represented by these 16 bits. If `nSequence[22]` is not set, then the relative lock time is k blocks. If it is set, then the relative lock time is $k \times 512$ seconds. The multiplier 512 was chosen because it is the power of two closest to 600 (the average number of seconds required to find a block). This choice allows the relative lock time to specify similar durations of time using either blocks or seconds. The maximum relative lock time in blocks is $2^{16} - 1 = 65,535$ blocks which corresponds to approximately 1.25 years. In terms of seconds, the maximum relative lock time is $(2^{16} - 1) \times 512 = 33,553,920$ seconds ≈ 1.06 years.

In spite of relying on the same `nSequence` field, absolute lock times (using `nLockTime`) and relative lock times can be enabled independently of each other. Both of them can be disabled by setting all the `nSequence` values to `0xFFFFFFFF`. Setting `nSequence[31]` to 0 in any transaction input enables both the absolute lock time for the whole transaction and the relative lock time for that particular input. Choosing `nSequence` strictly less than `0xFFFFFFFF` but with `nSequence[31]` equal to 1 in any transaction input enables the absolute lock time for the whole transaction and disables the relative lock time for that particular input. Finally, setting `nSequence[31]` to 0 and `nLockTime` to 0 effectively disables the absolute lock time for the whole transaction and enables the relative lock time for that particular input.

Output Format

Each output in a pre-SegWit regular transaction has the same three fields. Figure 5.2 shows these fields for the first output. The fields in the other outputs are not shown for brevity. The output fields have the following semantics.

- The 8-byte `nValue` field contains the number of satoshis (1 satoshi = 10^{-8} bitcoins) being locked in the output. For example, `nValue` = $2,500,000,000$ corresponds to 25 bitcoins.
- The `scriptPubkeyLen` field is a VarInt encoding of the length of the challenge script which is used to lock the output.
- The `scriptPubkey` field contains the challenge script itself.

5.3 Pre-SegWit Coinbase Transactions

A pre-SegWit coinbase transaction structurally looks like a pre-SegWit regular transaction with a single input and multiple outputs as shown in Figure 5.5. Like a regular transaction, the coinbase transaction begins with a 4-byte `nVersion` field that contains the transaction version number. The last field in the coinbase transaction is the 4-byte `nLockTime` field which is ignored as lock times on coinbase transactions are meaningless. Absolute lock times (via `nLockTime`) on regular transactions are intended to delay their inclusion in the blockchain until the latest block in the blockchain has a certain height or median-time-past value. When a miner creates a coinbase transaction for a new candidate block, he intends this transaction to be included in the next block on the blockchain. There is no reason for the miner to make the coinbase transaction invalid by imposing a lock time which expires sometime in the future.

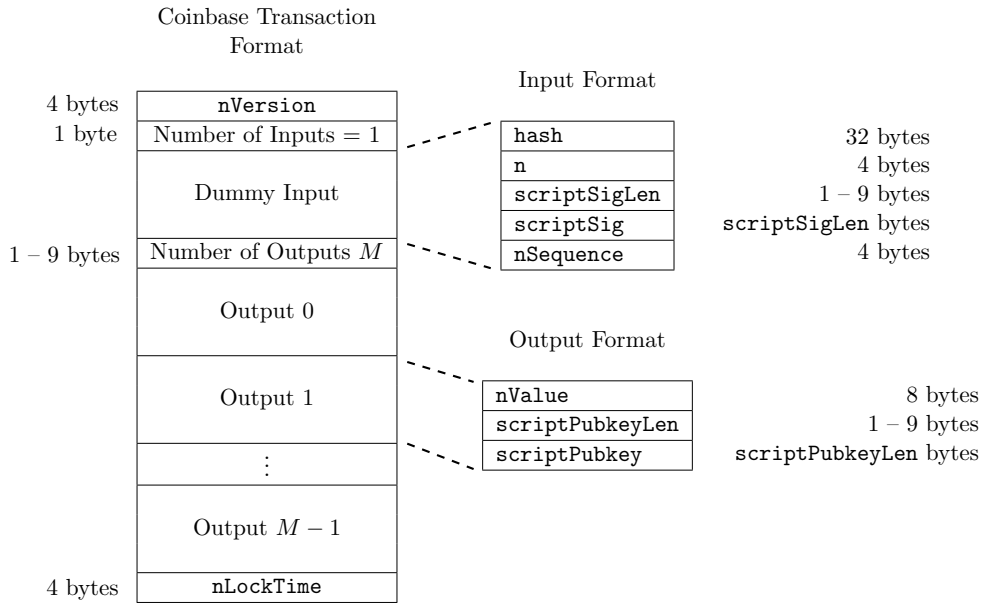


Figure 5.5: Pre-SegWit coinbase transaction format

Input Format

The sole input in the coinbase transaction is a dummy input in spite of having the five fields corresponding to a regular transaction input. It does not unlock a previous transaction output. The input fields have the following semantics.

- The 256-bit `hash` field is always set to all zeros (`0x00000...000`).
- The 4-byte `n` field is always set to `0xFFFFFFFF`.
- The `scriptSig` field does not contain a response script which can unlock a previous output. Instead, it contains the *coinbase* field which can have a maximum length of 100 bytes. The length of the coinbase field is specified in the `scriptSigLen` field.

  The height of the block containing the coinbase transaction is stored at the beginning of the coinbase field. The rest of the bytes can be arbitrarily set by the miner creating the coinbase transaction. Miners use these bytes to modify the `hashMerkleRoot` value in the block header in case they run out of `nNonce` values to try when mining the block.

  The height currently occupies the first four bytes in the coinbase field. The first byte is set to `0x03` to indicate that the height is encoded in the next three bytes. The following three bytes contain the height as a signed integer in little-endian format. When the block height increases

to a value that cannot be accommodated in three bytes (which will take more than 150 years), the first byte will be set to `0x04` and the following four bytes will be used to encode the height.

- The `nSequence` field is ignored as lock times are not relevant for coinbase transactions. Relative lock times do not make sense as the coinbase transaction input does not unlock a previous output. The non-applicability of absolute lock times was discussed a t the beginning of this section.

Why were the `hash`, `n`, `nSequence`, and `nLockTime` fields included in the coinbase transaction format if they either take fixed values or are ignored? While there is no way to know for sure, these fields were probably included to maintain consistency with the the regular transaction format. While wasteful in terms of space, this consistency translates to some minor conveniences in the C++ code of the Bitcoin Core reference client.

Output Format

The outputs in the coinbase transaction have the same fields as the outputs in a regular transaction. As in a regular transaction output, the `nValue` field specifies the amount of bitcoins (in satoshis) locked in the output, the `scriptPubkey` field contains a challenge script, and the `scriptPubkeyLen` field contains the length of `scriptPubkey` in bytes. The main difference is that the amount specified in the `nValue` field should be less than or equal to the block reward (block subsidy + transaction fees) in the block. Furthermore, the sum of the `nValue` fields from all the coinbase outputs should not exceed the block reward.

An output in a coinbase transaction cannot be spent until it has received 101 confirmations. So an input unlocking the coinbase transaction output of a block at height h has to be in a block at height strictly higher than $h + 100$. This rule ensures that coinbase transaction outputs are spent only after they are unlikely to become invalid due to a blockchain fork.

Why is the block height stored in the coinbase field? This is to ensure that the coinbase transactions in different blocks on the blockchain have different TXIDs. The coinbase transactions in two blocks mined by the same miner can contain the same challenge scripts in the `scriptPubkey` fields of the outputs. The `nValue` field values can also be the same. If the block height is not included in the coinbase field, then the `scriptSig` fields can also be the same. Consequently, the two coinbase transactions will have the same TXID (the double SHA-256 hash). This is undesirable as we will not be able to distinguish between the two coinbase transactions when we specify this TXID in the `hash` field of an input. The idea of including the block height in the coinbase field was proposed in BIP 34. This behaviour was made mandatory from the block at height 2,24,413 which was mined in March 2013.

5.4 Bitcoin Script

The challenge and response scripts stored in the `scriptPubkey` and `scriptSig` fields of a transaction are encoded using a scripting language which was developed specifically for Bitcoin. The language, which is simply called *Script*, is a stack-based language, i.e. it uses a stack to store input parameters and return values of a function. To execute a function which takes n arguments, the arguments are first pushed onto a stack. The function performs its calculation by reading these n values directly from the stack and stores the return value on the stack. The stack removes the need for variables to store the arguments or the return value.

Script has limited functionality when compared to general-purpose languages. For example, it does not support loops. But it does have functions which perform cryptographic operations specific to Bitcoin like SHA-256 hash calculation and ECDSA signature verification. There is no document defining the functionalities supported by Script. The implementation of the Script interpreter in the Bitcoin Core reference client serves as its de facto specification.

A Script script is just a bytestring which is parsed into a sequence of data values and operators. Operators are encoded in a single byte. For example, the byte `0x93` is used to represent the addition operator. The byte value is called the *opcode* of the operator. For the sake of exposition, operators are usually denoted by the names given to them in the source code of the Bitcoin Core reference client. These names start with the string `OP_`. For example, the addition operator is called `OP_ADD`.

The operators with opcodes in the range `0x00` to `0x60` (except for `0x50`) push data values of different lengths onto the stack as shown in Table 5.2. Each stack element is an array of bytes (an empty array is allowed). After the execution of an operator from Table 5.2, the number of elements in the stack increases by one and the top element contains the data value which was pushed onto the stack.

- The `OP_0` operator pushes an empty array of bytes onto the stack. An empty array is used to denote a Boolean return value of `False` from an operator. For this reason, the `OP_0` operator is also called `OP_FALSE`.

- The operators with opcodes in the range `0x01` to `0x4B` (1 to 75 in decimal) are used to push upto 75 bytes following the operator onto the stack. These operators do not have names starting with `OP_` in the Bitcoin Core reference client source code.

- The operators `OP_PUSHDATA1`, `OP_PUSHDATA2`, and `OP_PUSHDATA4` enable the storage of data values having length greater than 75 bytes on the stack. While the `OP_PUSHDATA4` operator can specify the push of upto $2^{32} - 1$ bytes onto the stack, the Bitcoin protocol limits the largest size

Opcode	Operator Name	Operator Action
0x00	OP_0	Push an empty array of bytes onto the stack.
0x01	Not applicable	Push the next 0x01 bytes onto the stack.
0x02	Not applicable	Push the next 0x02 bytes onto the stack.
⋮	⋮	⋮
0x4B	Not applicable	Push the next 0x4B bytes onto the stack.
0x4C	OP_PUSHDATA1	Let N be integer represented by the single byte b immediately following OP_PUSHDATA1. Push the N bytes after byte b onto the stack.
0x4D	OP_PUSHDATA2	Let N be integer represented by the two bytes b_1b_2 immediately following OP_PUSHDATA2. Push the N bytes after bytes b_1b_2 onto the stack.
0x4E	OP_PUSHDATA4	Let N be integer represented by the four bytes $b_1b_2b_3b_4$ immediately following OP_PUSHDATA4. Push the N bytes after bytes $b_1b_2b_3b_4$ onto the stack.
0x4F	OP_1NEGATE	Push the number -1 onto the stack.
0x51	OP_1	Push the number 1 onto the stack.
0x52	OP_2	Push the number 2 onto the stack.
⋮	⋮	⋮
0x60	OP_16	Push the number 16 onto the stack.

Table 5.2: Script operators which push data onto the stack

of a data value which can be pushed to 520 bytes. Scripts containing data pushes of more than 520 bytes are considered invalid.

- The operator OP_1NEGATE pushes the number -1 onto the stack. The operators OP_1 to OP_16 are used to push numbers in the range 1 to 16 onto the stack. While these push operations can be performed using the opcode 0x01, doing so will require two bytes: one byte for the opcode 0x01 and one byte for the number to be pushed. Specific operators for pushing the numbers $-1, 1, 2, \ldots, 16$ were probably included because these small numbers are more likely to occur in scripts.

- The opcode 0x50 is reserved for future use. There is a minor convenience in skipping 0x50 and using 0x51 as the opcode for OP_1. The number to be pushed by the operators OP_1NEGATE, OP_1, ..., OP_16 can be expressed as the difference between their opcodes and 0x50. For example, $-1 =$ 0x4F $-$ 0x50 and $1 =$ 0x51 $-$ 0x50.

Opcode	Operator Name	Operator Action
0x99	OP_IF	If the top stack element is True, execute the statements until the next OP_ELSE or OP_ENDIF. The top stack element is popped from the stack.
0x76	OP_DUP	Pushes a copy of the top stack element onto the stack
0x93	OP_ADD	Pops the top two stack elements and pushes their sum onto the stack
0xAA	OP_HASH256	Pops the top stack element and pushes its double SHA-256 hash onto the stack.

Table 5.3: Examples of Script operators

The operators with opcodes in the range 0x61 to 0xB2 (97 to 178 in decimal) specify operators which perform flow control, stack manipulation, arithmetic, and cryptographic operations.[4] Table 5.3 lists some examples of such operators. A complete list of all the operators and their definitions can be found in the Bitcoin Wiki.[5] We will describe some of these operators in the next section.

Script uses postfix notation to express operations which are not the data push operations listed in Table 5.2. In postfix notation, the parameters of an operator are specified before the operator. For example, the postfix notation for the sum 2 + 3 is 2 3 +. In Script, this postfix expression would be given by the bytestring 0x525393 which corresponds to OP_2 OP_3 OP_ADD when the opcodes are replaced with operator names. Figure 5.6 shows the state of the stack when different parts of the expression are executed. We assume that the stack is initially empty. The expression is evaluated from left to right. The OP_2 operator is executed first resulting in the number 2 being pushed onto the stack. The execution of OP_3 pushes the number 3 onto the stack. When OP_ADD is executed, the numbers 3 and 2 are popped off the stack and their sum 5 is pushed onto the stack.

Challenge and Response Script Execution

A transaction input can unlock an unspent transaction output (UTXO) for spending by providing a response script in its scriptSig field which contains a valid response to the challenge script stored in the UTXO's scriptPubkey field. When a new transaction is broadcast on the network for inclusion in the blockchain, the nodes which receive it verify that the transaction inputs

[4]Not all the opcodes in the range 0x61 to 0xB2 correspond to operators. This is because some operators which were defined in the original implementation of the Bitcoin client by Satoshi Nakamoto were later removed due to security concerns.

[5]https://en.bitcoin.it/wiki/Script

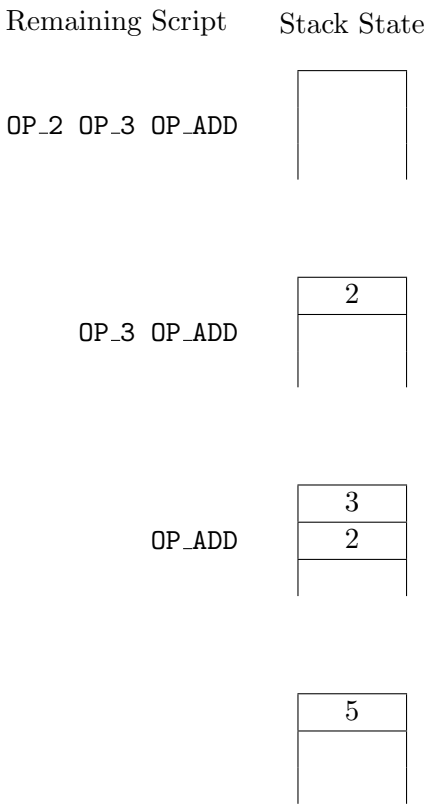


Figure 5.6: Stack state during the execution of `OP_2 OP_3 OP_ADD`

contain valid response scripts. If any of the response scripts are invalid, the nodes will reject the transaction and not re-broadcast it to their neighbors. The nodes validate a response script in the following manner:

1. The response script is first executed using an empty stack. If the response script execution terminates with an error, it is considered invalid.

2. If the response script execution succeeds, the state of the stack at the end of the execution is used to execute the challenge script. If the challenge script execution terminates with an error, the response script is considered invalid.

3. If the challenge script execution succeeds, the response script is considered valid if the stack at the end of the execution is not empty and the top stack element evaluates to `True`. A stack element evaluates to `False` if it is either an empty array of bytes or the little-endian encoding of the integer 0 (signed or unsigned). Otherwise, it evaluates to `True`.

The above steps are illustrated in Figure 5.7. Before response script execution, the stack is empty. After the response script is executed, the stack contains

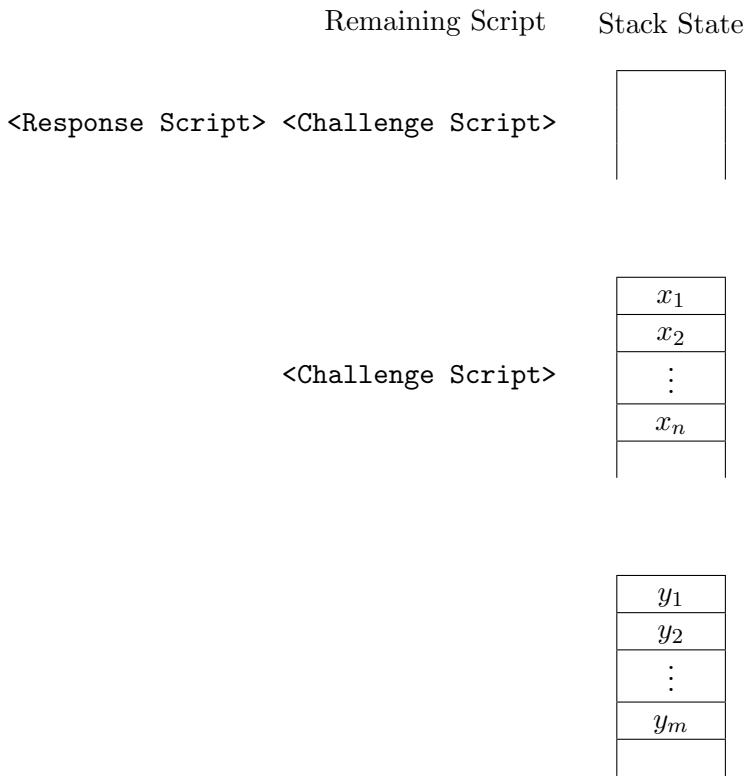


Figure 5.7: Stack state during the execution of the response and challenge scripts

the items $x_1, x_2, \ldots, x_n$. This stack state is used to execute the challenge script resulting in the stack containing the items $y_1, y_2, \ldots, y_m$. The response script is considered valid if y_1 evaluates to `True`. A valid response script populates the stack with items which ensure that the subsequent challenge script execution proceeds without errors and ends with a top stack element that evaluates to `True`.

As an example, consider the challenge script:

```
OP_HASH256 0x20 <256-bit string> OP_EQUAL
```

For convenience, let `S` denote the 256-bit string appearing in the above script. Figure 5.8 shows the state of the stack during the execution of this script. The top stack element is assumed to be x before script execution. The `OP_HASH256` operator pops the top stack element x and pushes its double SHA-256 hash $H(x)$ onto the stack. The `0x20` operator pushes a 32-byte array containing `S` onto the stack. The `OP_EQUAL` operator pops the top two stack elements and pushes the number 1 onto the stack if $H(x)$ and `S` are equal. If they are not equal, it pushes the number 0 onto the stack.

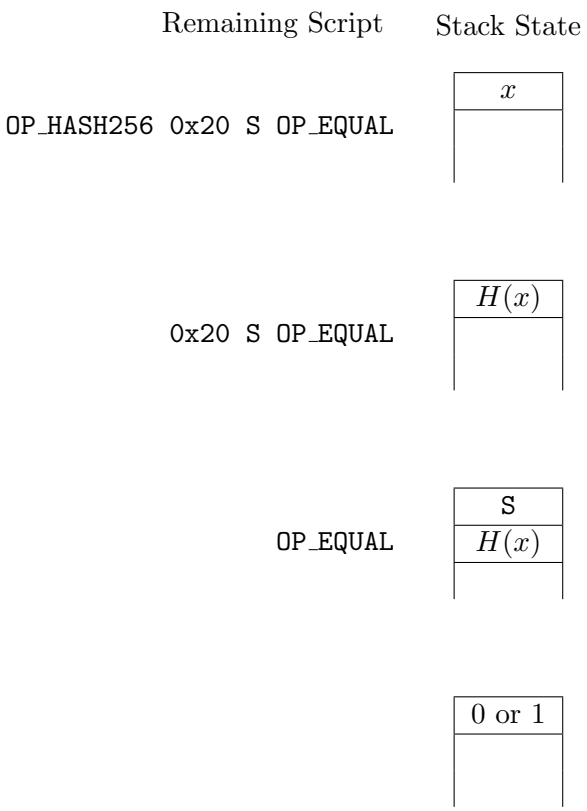


Figure 5.8: Stack state during the execution of the challenge script `OP_HASH256 0x20 S OP_EQUAL`

As 1 evaluates to `True` and 0 evaluates to `False`, the challenge script effectively asks for a preimage of `S` under the double SHA-256 hash function. A valid response script consists of a single data push operator which pushes a preimage x of `S` onto the stack.

While the challenge script used in this example is valid Script code, it does not belong to the set of standard challenge scripts which are discussed in the next section. However, it can be embedded inside a type of standard challenge script called pay-to-script-hash (P2SH).

5.5 Pre-SegWit Standard Scripts

When a new regular transaction is broadcast on the network for inclusion in the blockchain, each node which hears the transaction validates it by evaluating the challenge and response scripts. Valid transactions are then relayed by the node to its neighboring nodes, which in turn will perform their own validation. If no restrictions are placed on the challenge and response scripts, a malicious node can construct scripts which consume excessive amounts of

CPU time or RAM at each network node during script validation. This constitutes a denial-of-service (DoS) attack on the network as regular transactions broadcasted by non-malicious nodes will experience delays before they are recorded on the blockchain. To avoid such DoS attacks, network nodes will not relay transactions containing scripts which do not belong a limited set of *standard scripts*. Prior to SegWit activation, the set of standard challenge scripts consisted of the five script templates: Pay to Public Key (P2PK), Pay to Public Key Hash (P2PKH), m-of-n Multi-signature, Pay to Script Hash (P2SH), and Null Data.[6]

Pay To Public Key (P2PK)

The P2PK challenge script template has a `scriptPubkey` field which consists of a data push of a public key followed by the `OP_CHECKSIG` operator. The public key can be in either compressed (33 bytes) or uncompressed (65 bytes) format. For compressed public keys, the `scriptPubkey` is of the form

```
0x21 <Compressed Public Key> OP_CHECKSIG
```

where the `0x21` operator pushes a byte array containing the next 33 bytes onto the stack. For uncompressed public keys, the `scriptPubkey` has the form

```
0x41 <Uncompressed Public Key> OP_CHECKSIG
```

where the operator `0x41` pushes a byte array containing the next 65 bytes onto the stack. Usually, the sizes of data push operations are omitted from script descriptions. With such an omission, the P2PK challenge script template is given by

```
<Public Key> OP_CHECKSIG.
```

Let x_1 and x_2 be the top two elements in the stack when the `OP_CHECKSIG` operator is executed. `OP_CHECKSIG` uses the ECDSA signature verification procedure from Section 2.5 to check that x_2 is a valid signature when x_1 is used as the public key. Some fields from the transaction containing the `scriptPubkey` are used as the message m for the signature verification (see Section 5.6). `OP_CHECKSIG` pushes an empty array of bytes (which evaluates to `False`) onto the stack if the signature is invalid. For valid signatures, a single non-zero byte (which evaluates to `True`) is pushed onto the stack.

A valid response to the P2PK challenge script contains a single data push of a byte array containing a valid signature. So the `scriptSig` field is of the form

```
<Signature>
```

where we have omitted the data push operator which pushes `<Signature>` onto the stack. While the ECDSA with `secp256k1` domain parameters generates a signature which is 64 bytes long, the `scriptSig` field contains a variable

[6] SegWit added two more script templates to this set which are described in Section 5.8.

length encoding of this signature according to the Distinguished Encoding Rules (DER) of the X.690 standard.[7] The DER encoding of the ECDSA signatures does not add any value to the Bitcoin protocol. It is present because the original implementation of the Bitcoin Core client used the OpenSSL[8] library for creating and validating ECDSA signatures. OpenSSL uses DER encoding to represent ECDSA signatures. In 2016, the dependence of Bitcoin Core on OpenSSL was removed and a new library called `libsecp256k1` was added to perform the ECDSA signature generation and validation. To maintain compatibility with the earlier signature format, `libsecp256k1` continues to use the DER encoding. A single byte encoding the *signature hash type* is appended to the DER encoding of the ECDSA signature. The signature hash type indicates which parts of the transaction containing the `scriptPubkey` are included in the message m which is used to generate the signature. Signature hash types will be discussed in Section 5.6. Including the data push operator, the final length of the `scriptSig` field containing a valid signature is at most 74 bytes.

Figure 5.9 shows the state of the stack during the execution of the P2PK response and challenge scripts. Recall that the response script is first executed using an empty stack followed by the challenge script execution. The response script pushes `<Signature>` onto the stack. The challenge script pushes `<Public Key>` onto the stack and executes the `OP_CHECKSIG` operator. This operator pops the top two stack elements and pushes a `True` value onto the stack if the signature is valid. If the signature is invalid, a `False` value is pushed onto the stack.

A response script is considered a valid response to a challenge script if the top stack element at the end of the challenge script execution evaluates to `True`. In the P2PK case, the response script has to contain a valid ECDSA signature created by the private key corresponding to the public key in the challenge script.

As discussed in Chapter 2, there is no known method to recover the private key from the public key in a computationally feasible manner. So the public key can be safely revealed and used as a receiving address for bitcoin payments via the P2PK script template. Public keys used in this manner are called P2PK addresses. In practice, P2PKH addresses are used instead of P2PK addresses for better security (see Section 3.3).

As the P2PK challenge script contains a public key and the P2PK response script contains a signature, the names `scriptPubkey` and `scriptSig` for the variables containing these scripts in the Bitcoin Core client were probably chosen keeping the P2PK script template in mind. Sometimes challenge scripts are called *pubkey scripts* and response scripts are called *signature scripts*. We will see that these names do not accurately describe the contents of the

[7] See `https://en.wikipedia.org/wiki/X.690`

[8] See `https://en.wikipedia.org/wiki/OpenSSL`

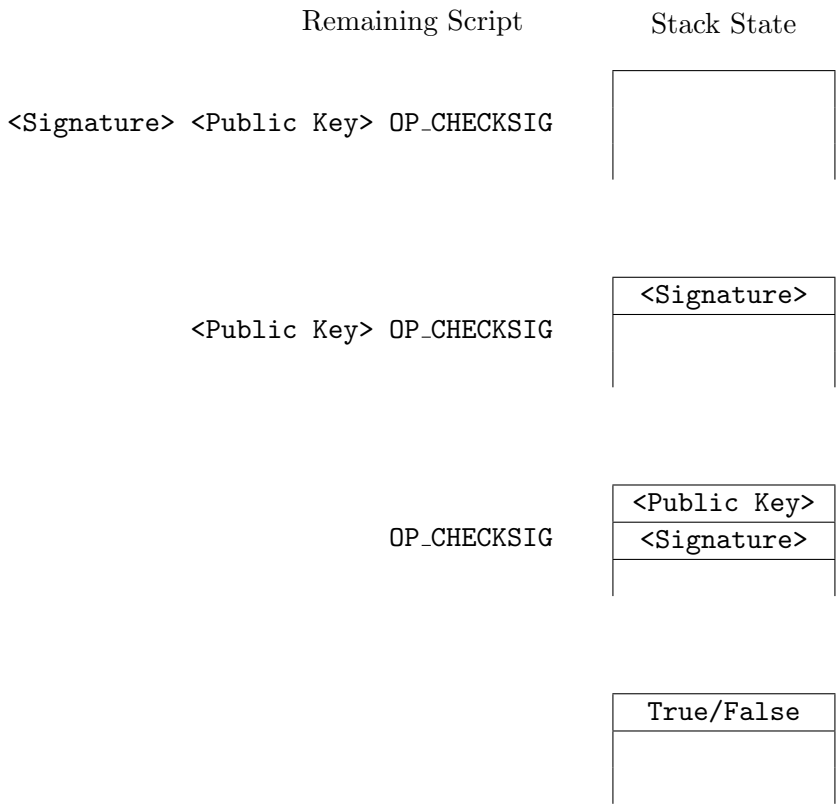


Figure 5.9: Stack state during the execution of P2PK response and challenge scripts

challenge and response scripts in the other script templates.

Pay To Public Key Hash (P2PKH)

The creation of P2PKH addresses was described in Section 3.3. These addresses are derived from an uncompressed public key `0x04`$\|X\|Y$ as shown in Figure 3.2. They are represented using an alphanumeric string in Base58 format. While the public key itself cannot be recovered from a P2PKH address, the SHA-256 + RIPEMD-160 hash R of the public key can be recovered as shown in Figure 5.10. In the figure, B represents a single byte containing the address version and C_4 represents the 4-byte checksum.

The P2PKH challenge script template is given by

`OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`.

where `<PubKeyHash>` represents the SHA-256 + RIPEMD-160 hash R of an uncompressed public key. The `OP_CHECKSIG` operator works as in a P2PK challenge script. The other operators in the script work as described below.

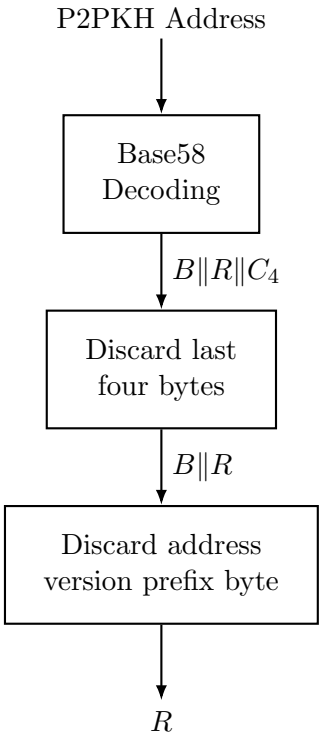


Figure 5.10: Recovering the hash R = RIPEMD-160(SHA-256(`0x04`$\|X\|Y$)) of an uncompressed public key `0x04`$\|X\|Y$ from its P2PKH address

- The `OP_DUP` operator duplicates the top stack element, i.e. it pushes a copy of the top stack element onto the stack.
- The `OP_HASH160` operator pops the top stack element x and pushes RIPEMD-160(SHA-256(x)) onto the stack.
- The `OP_EQUALVERIFY` operator pops the top two stack elements and compares them. If they are equal, the script execution continues. If they are not equal, the script terminates with an error.

A valid response to the P2PKH challenge script contains exactly two data pushes: the first one pushes a byte array containing a valid signature and the second one pushes a byte array containing an uncompressed public key. So the `scriptSig` field is of the form

`<Signature> <Public Key>`.

Figure 5.11 shows the state of the stack during the execution of the P2PKH response and challenge scripts. The execution proceeds as follows:

1. The response script pushes `<Signature>` and `<Public Key>` onto the stack.

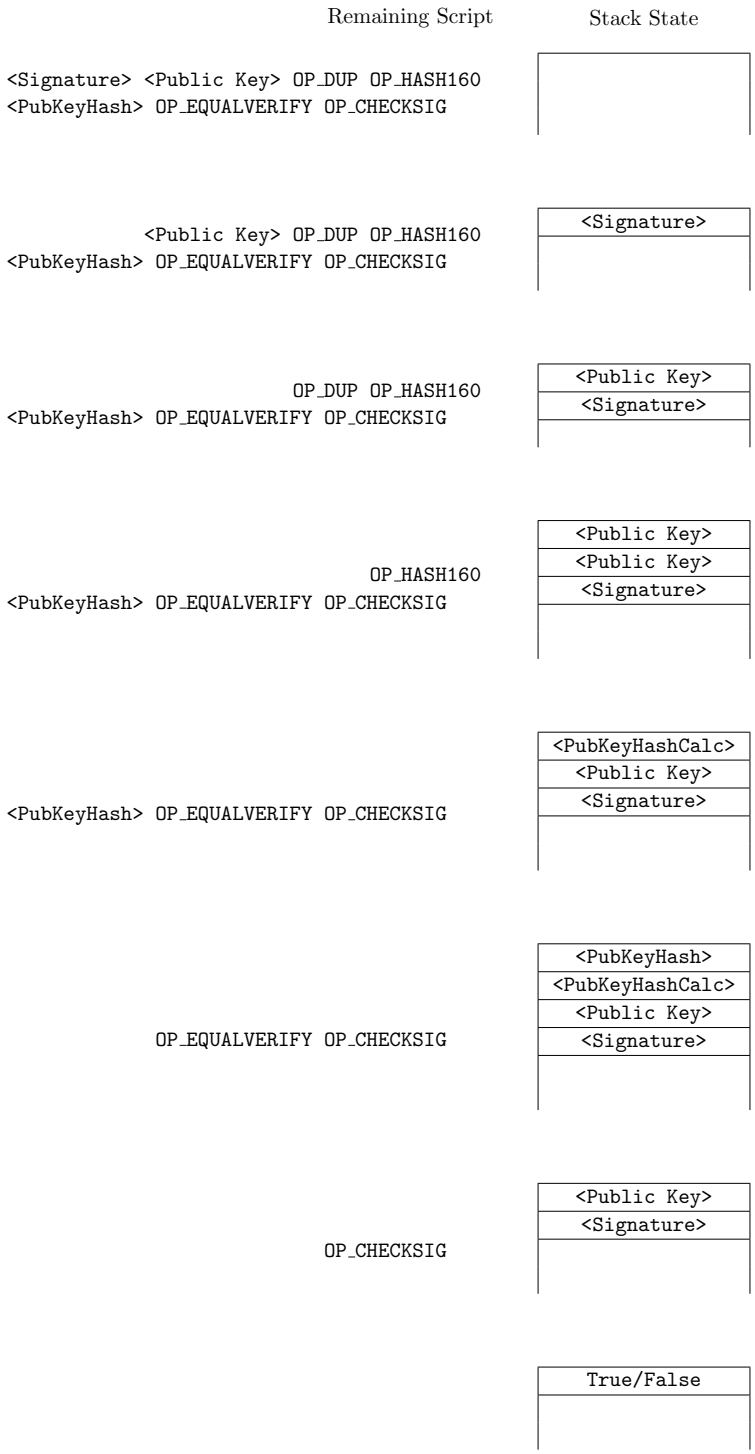


Figure 5.11: Stack state during the execution of P2PKH response and challenge scripts

2. The `OP_DUP` operator in the challenge script pushes a copy of the top stack element `<Public Key>` onto the stack.

3. The `OP_HASH160` operator pops the top stack element `<Public Key>` and calculates its SHA-256 + RIPEMD-160 hash `<PubKeyHashCalc>`. This hash is then pushed onto the stack.

4. The public key hash `<PubKeyHash>` from the challenge script is pushed onto the stack.

5. The `OP_EQUALVERIFY` operator pops and compares the top two stack elements `<PubKeyHash>` and `<PubKeyHashCalc>`. If they are equal, then the script execution proceeds. If they are not equal, the script execution terminates with an error. Equality implies that the `<Public Key>` provided by the response script has a SHA-256 + RIPEMD-160 hash which is equal to the `<PubKeyHash>` given in the challenge script.

6. If the script execution has not terminated, the `OP_CHECKSIG` operator checks that the `<Signature>` provided by the response script is a valid signature using the `<Public Key>`. It pushes a `True` value onto the stack if the signature is valid. If the signature is invalid, a `False` value is pushed onto the stack.

The values in the response script can cause the challenge script execution to fail in two[9] ways:

- The hash `<PubKeyHashCalc>` calculated from the `<Public Key>` given in the response script is not equal to the hash `<PubKeyHash>` given in the challenge script.

- The `<Signature>` given in the response script fails the `secp256k1` ECDSA signature validation procedure performed using the `<Public Key>` given in the response script.

In order to provide a valid response script to a P2PKH challenge script, the entity providing the response has to first know the public key whose hash is `<PubKeyHash>`. Furthermore, it has to also know the private key corresponding to that public key to be able to create a valid signature.

As discussed in Section 3.3, the main advantage of P2PKH addresses over P2PK addresses is that an adversary looking to derive the private key from a P2PKH address has to first solve the difficult problems of finding RIPEMD-160 and SHA-256 preimages. But once an output containing a P2PKH challenge script in its `scriptPubkey` field is unlocked by an input containing a valid response script, the public key corresponding to the P2PKH address is

[9]Response and challenge script execution can also fail if the data push operations try to push more than 520 bytes onto the stack. We ignore such failure scenarios.

revealed as it is contained in the `scriptSig` field of the input. This revelation occurs as soon as the transaction containing the input is broadcast on the network and becomes permanent once this transaction is recorded on the blockchain. While an adversary with access to the public key has to still solve an instance of the difficult ECDLP to obtain the private key, the extra layers of protection provided by the hash functions are lost. For this reason, it is recommended to not reuse the same P2PKH address in multiple Bitcoin transactions. It is better to create a unique P2PKH address for each transaction. The total number of available P2PKH addresses is equal to the number of private keys associated with the `secp256k1` domain parameters (see Section 2.4). The number of private keys is $n - 1$ where n is the 256-bit integer given in equation (2.3). As $n - 1$ is approximately 1.157×10^{77}, there is no risk of running out of P2PKH addresses by using a new one for each transaction.

m-of-n Multi-Signature (Multisig)

A m-of-n multisig challenge script specifies n public keys and requires a valid response script to provide m ECDSA signatures created using *any* m out of the n private keys corresponding to these public keys. Multisig challenge scripts enable various forms of joint ownership of the bitcoins associated with an output. Consider the following examples where Alice knows the private key `<PrivKeyA>` corresponding to a public key `<PubKeyA>`, Bob knows the private key `<PrivKeyB>` corresponding to a public key `<PubKeyB>`, and Carol knows the private key `<PrivKeyC>` corresponding to a public key `<PubKeyC>`.

- Suppose $m = n = 2$. Let the 2-of-2 multisig challenge script specify the public keys `<PubKeyA>` and `<PubKeyB>`. A valid response script requires two valid ECDSA signatures created using the private keys `<PrivKeyA>` and `<PrivKeyB>`. Such a response script cannot be created by Alice or Bob alone and requires both of them to provide their respective signatures. Thus the output cannot be spent unless both Alice and Bob agree to spend it.

- Suppose $m = 1$ and $n = 3$. Let the 1-of-3 multisig challenge script specify the three public keys `<PubKeyA>`, `<PubKeyB>`, and `<PubKeyC>`. But a valid response script requires only one valid ECDSA signature created using any one of the three private keys `<PrivKeyA>`, `<PrivKeyB>`, `<PrivKeyC>`. Thus the output can be spent by any one of Alice, Bob or Carol.

- Suppose $m = 2$ and $n = 3$. Let the 2-of-3 multisig challenge script specify the three public keys `<PubKeyA>`, `<PubKeyB>`, and `<PubKeyC>`. A valid response script requires only two valid ECDSA signatures created using any one two of the three private keys `<PrivKeyA>`, `<PrivKeyB>`,

<PrivKeyC>. Thus the output can be spent if any two of Alice, Bob and Carol agree to spend it.

The scriptPubkey of the m-of-n challenge script is of the form

m <Public Key 1> ··· <Public Key n> n OP_CHECKMULTISIG

where the OP_CHECKMULTISIG operator checks that the signatures provided by the response script are valid. A valid response script pushes m signatures onto the stack. The scriptSig field is of the form

OP_0 <Signature 1> ··· <Signature m>.

where the OP_0 operator which pushes an empty array onto the stack is present to account for a bug in the OP_CHECKMULTISIG operator implementation. The bug causes OP_CHECKMULTISIG to pop one extra item off the stack. This bug cannot be fixed without requiring all the nodes in the network to upgrade their client software. If some of the nodes do not upgrade their clients, it would result in a *hard fork* in the Bitcoin blockchain due to upgraded and non-upgraded nodes disagreeing on the validity of the multisig challenge scripts.[10]

Figure 5.12 shows the state of the stack during the execution of the m-of-n multisig response and challenge scripts. For brevity, some of the intermediate stack states consisting of only data push operations have been omitted from the figure. The execution proceeds as follows:

1. The OP_0 operator in the response script pushes an empty byte array onto the stack. The m signatures <Signature 1>, ..., <Signature m> are then pushed onto the stack.

2. The challenge script pushes the integer m, the n public keys <Public Key 1>, ..., <Public Key n> onto the stack, and integer n onto the stack.

3. The OP_CHECKMULTISIG operator obtains the number of public keys provided by reading the top stack element n. It then obtains the number of signatures provided by reading the element m. If the signatures <Signature 1>, ..., <Signature m> were created by a sequence of m private keys whose corresponding public keys form a subsequence[11]of <Public Key 1>, ..., <Public Key n>, then the OP_CHECKMULTISIG operator pops the top $m+n+3$ elements from the stack and pushes a True value onto the stack. Otherwise, it pops the top $m+n+3$ elements from the stack and pushes a False value onto the stack.

[10]See Chapter 7 for details about why hard forks are undesirable.

[11]A subsequence of a ordered sequence of elements $x_1, x_2, \ldots, x_n$ is given by $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, i.e. some elements can be omitted from the original sequence but the order of the remaining elements is unchanged.

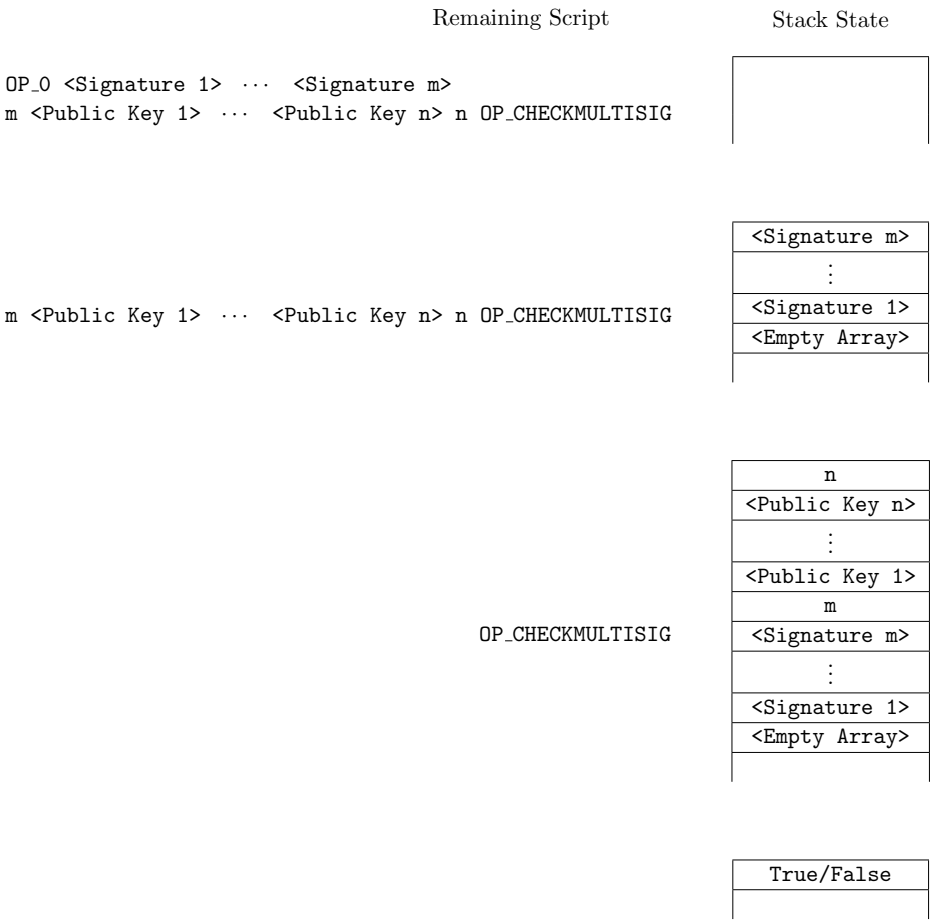


Figure 5.12: Stack state during the execution of m-of-n multisig response and challenge scripts

The subsequence restriction requires that the m signatures in the response script appear in the same order as their corresponding public keys in the challenge script. For example, suppose a 2-of-3 multisig challenge script is given by

`OP_2 <PubKeyA> <PubKeyB> <PubKeyC> OP_3 OP_CHECKMULTISIG`

where Alice, Bob, and Carol control the private keys corresponding to the public keys `<PubKeyA>`, `<PubKeyB>`, and `<PubKeyC>` respectively. Suppose `<SigA>`, `<SigB>`, and `<SigC>` are signatures created by Alice, Bob, and Carol respectively. Then the following response script is valid because the signatures from Alice and Carol appear in the same order as their public keys in the challenge script.

`OP_0 <SigA> <SigC>`.

But the following response script is invalid because the signatures from Alice and Carol do not appear in the same order as their public keys in the challenge script.

```
OP_0 <SigC> <SigA>.
```

While the `OP_CHECKMULTISIG` operator allows n to be as large as 20, the Bitcoin protocol considers m-of-n multisig challenge scripts with n greater than 3 to be non-standard. But m-of-n multisig scripts with values of n upto 15 are considered standard if these scripts are embedded inside a P2SH challenge script. To distinguish between the two types of multisig scripts, the former version is called a *bare multisig* script while the latter is called a *P2SH multisig* script.

Pay To Script Hash (P2SH)

The P2SH challenge script was added to the list of standard scripts in 2012 to bring the usability advantages of P2PKH challenge scripts to arbitrary challenge scripts (including m-of-n multisig scripts). Suppose Alice is requesting a bitcoin payment from Bob and wants the output containing the payment to be locked by a P2PKH challenge script. All Alice needs to do is provide the corresponding P2PKH address to Bob and he can create the transaction containing the payment. The P2PKH address can be encoded in a QR code[12] which can be scanned by Bob using a mobile phone camera. Or Alice can send the address to Bob via email. If Bob makes a mistake in typing the P2PKH address, the checksum in the P2PKH address will help detect the mistake. These advantages are not present if Alice wants the output containing the payment to be locked by an arbitrary challenge script. Unlike a P2PKH address which is at most 34 characters in Base58 format, an arbitrary challenge script may be too long to fit in a single QR code image. There is also no checksum to prevent typing errors. Furthermore, a large challenge script will require Bob to pay more transaction fees to make the payment as the fees are proportional to the transaction size. For example, a P2PKH challenge script is 25 bytes long while a 2-of-3 multisig challenge script is 105 bytes long. So if Alice requests payment to the latter script, Bob would have to pay $80r$ satoshis more as transaction fees where r is the fees per byte in satoshis. If Alice is a merchant accepting bitcoin as payment, she may not want to inconvenience her customer Bob with the increased transaction fees caused by payments to arbitrary challenge scripts.

The P2SH challenge script template retains the advantages of P2PKH challenge scripts while allowing the use of more complex challenge scripts to lock an output. It has a 23-byte `scriptPubkey` field of the form

```
OP_HASH160 <RedeemScriptHash> OP_EQUAL
```

[12] `https://en.wikipedia.org/wiki/QR_code`

where `<RedeemScriptHash>` represents the SHA-256 + RIPEMD-160 hash of a script called the *redeem script*. The redeem script is itself a challenge script which is specified in the `scriptSig` field along with a valid response to it. The `scriptSig` field of a P2SH response script has the form

```
<Response To Redeem Script> <Redeem Script Byte Array>
```

where `<Redeem Script Byte Array>` is a data push of the entire redeem script `<Redeem Script>` as a byte array onto the stack as a single item. The `<Response To Redeem Script>` portion of the `scriptSig` field contains a valid response to the redeem script `<Redeem Script>`.

Figure 5.13 shows the state of the stack during the execution of the P2SH response and challenge scripts. We assume that the `<Response To Redeem Script>` portion of the response script consists of data push operations pushing data $x_n, x_{n-1}, \ldots, x_1$ onto the stack. The execution proceeds as follows:

1. The `<Response To Redeem Script>` portion of the P2SH response script populates the stack with data items $x_1, x_2, \ldots, x_n$.

2. The redeem script specified by the byte array `<Redeem Script Byte Array>` is pushed onto the stack. The state of the stack at this point is saved for later use.

3. The `OP_HASH160` operator pops the top stack element `<Redeem Script>` and calculates its SHA-256 + RIPEMD-160 hash `<RedeemScriptHashCalc>`. This hash is then pushed onto the stack.

4. The redeem script hash `<RedeemScriptHash>` from the challenge script is pushed onto the stack.

5. The `OP_EQUAL` operator pops and compares the top two stack elements `<RedeemScriptHash>` and `<RedeemScriptHashCalc>`. If they are equal, then the number 1 is pushed onto the stack and the script execution continues. If they are not equal, the number 0 is pushed onto the stack and the script execution terminates with an error. Equality implies that the redeem script provided by the response script has a SHA-256 + RIPEMD-160 hash which is equal to the `<RedeemScriptHash>` given in the challenge script.

6. If the top stack element is 1, then the state of the stack which was saved in step 2 is restored. The redeem script `<Redeem Script>` is popped from the stack and executed. If the top stack element evaluates to `True` after the redeem script execution, then the P2SH response script specified in the `scriptSig` field is considered valid. Otherwise, it is considered invalid.

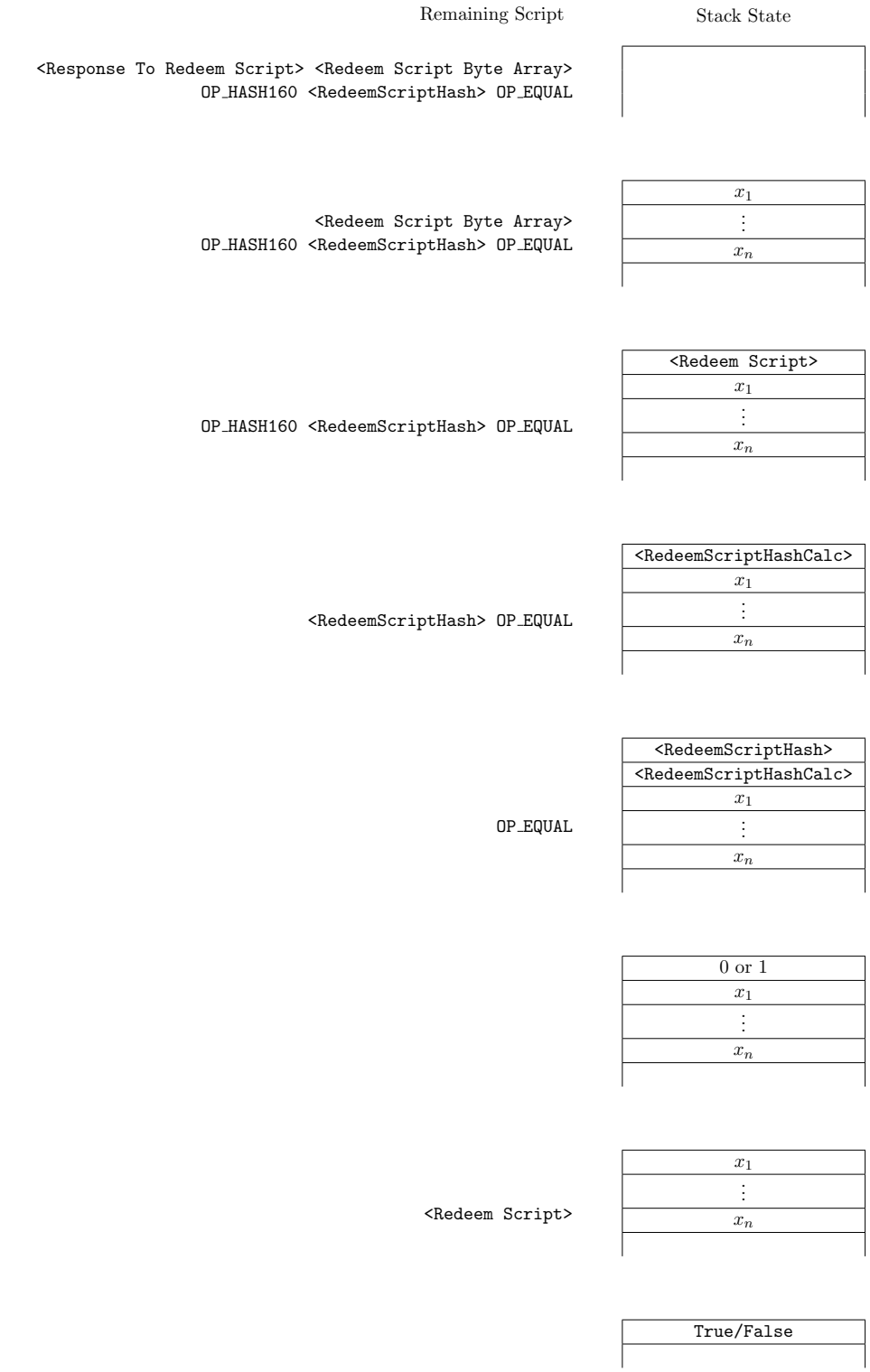


Figure 5.13: Stack state during the execution of P2SH response and challenge scripts

As an example, let us consider the case when the redeem script is a 2-of-3 multisig challenge script. The form of the `scriptPubkey` field is unchanged from the general case. The `scriptSig` field is given by

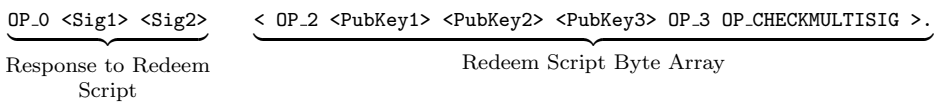


Figure 5.14 shows the state of the stack during the execution of the P2SH 2-of-3 multisig response and challenge scripts. Figure 5.14(a) shows the execution until the beginning of the redeem script execution and Figure 5.14(b) shows the redeem script execution. The last state in the former figure is repeated as the first state in the latter figure for continuity. In Figure 5.14(a), we have omitted the intermediate states showing the data pushes of the empty byte array (by `OP_0`) and `<Sig1>` for brevity. After `<Sig2>` is pushed onto stack, the entire redeem script byte array is pushed onto the stack as a single item. In the redeem script execution shown in Figure 5.14(b), the operators in the redeem script are executed. The redeem script is enclosed in angle brackets `<...>` to differentiate the data push of the redeem script as a byte array from its execution.

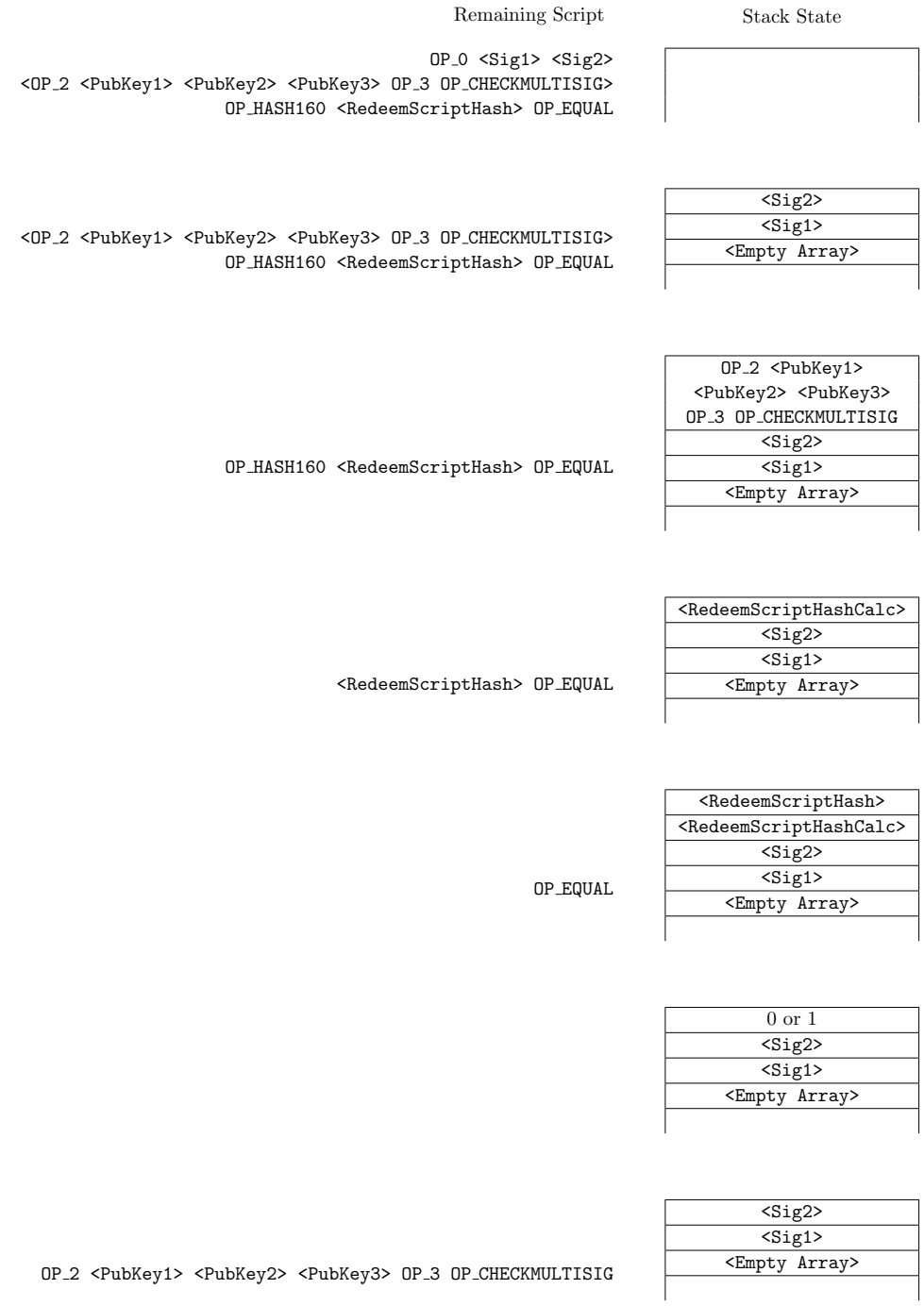
Suppose Alice wants Bob to make a bitcoin payment to an output which can be unlocked by providing signatures created by any two out of three private keys. She can of course share the three public keys required to create the 2-of-3 bare multisig challenge script with Bob. Alternatively, she can specify this challenge script as the redeem script in the P2SH script template and send the SHA-256 + RIPEMD-160 hash of the redeem script to Bob. This hash can be conveniently shared with Bob using a *P2SH address* which is similar to the P2PKH address described in Section 3.3. The generation of a P2PKH address from a public key is illustrated in Figure 3.2. The P2SH address generation procedure is essentially the same except for the following two differences.

1. Instead of the uncompressed public key, the redeem script is hashed first with SHA-256 and then with RIPEMD-160.

2. The address version byte which is prefixed to the hash is `0x05` for mainnet addresses and `0xC4` for testnet addresses.

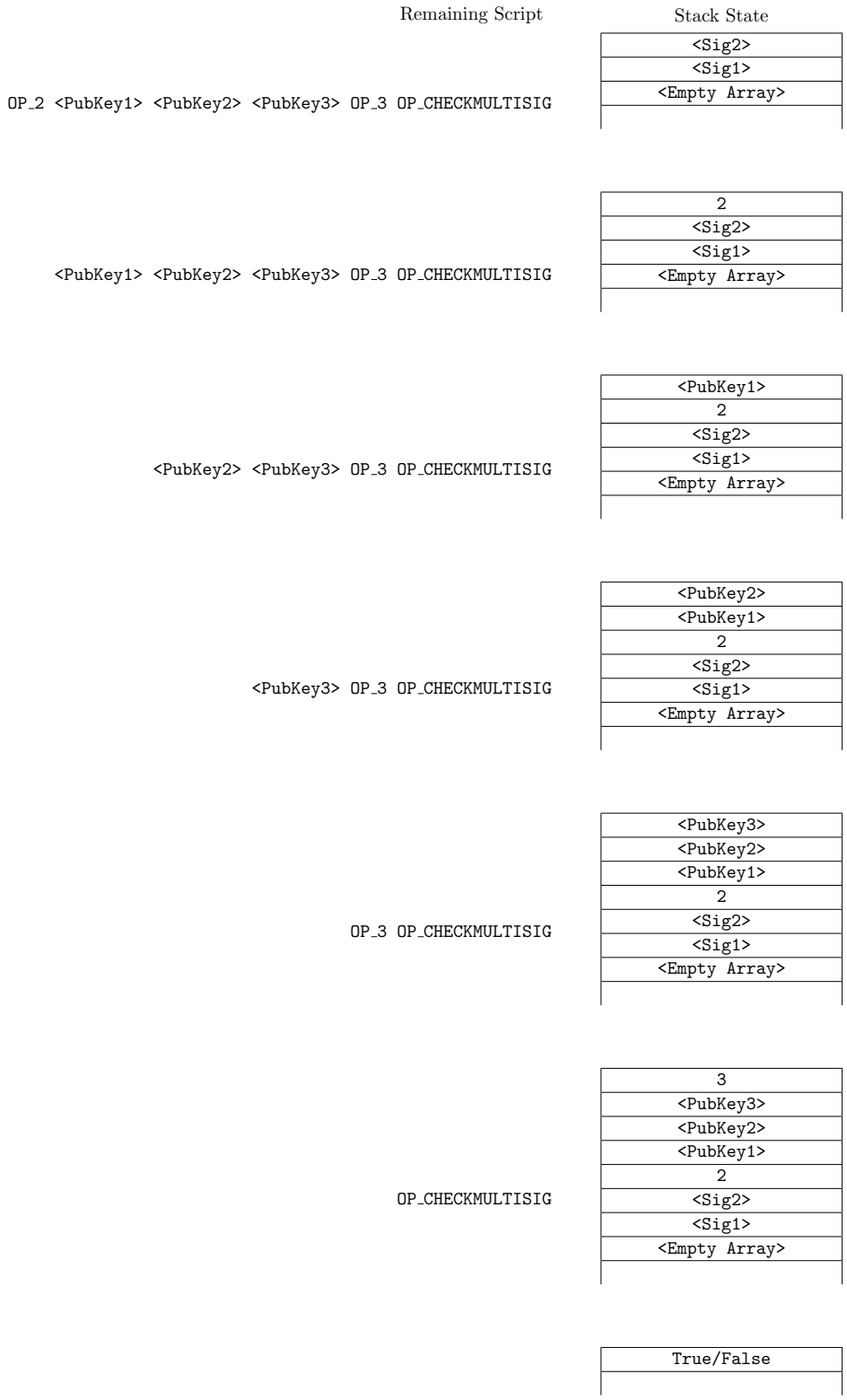
The checksum calculation and the Base58 encoding procedures are the same as in the P2PKH address generation.

As the address version byte for P2SH addresses on mainnet is `0x05`, the input to the Base58 encoding procedure is a number described by 25 hexadecimal digits in the range `0x050000...0000` to `0x05FFFF...FFFF`. All the numbers in this range lie between 2×58^{33} and $2 \times 58^{33} + 25 \times 58^{32}$. Hence they all begin with the number 2 and consist of exactly 34 digits in base 58



(a) P2SH multisig execution until the beginning of redeem script execution

Figure 5.14: Stack state during the execution of 2-of-3 P2SH multisig response and challenge scripts



(b) Redeem script execution

Figure 5.14: Stack state during the execution of 2-of-3 P2SH multisig response and challenge scripts (continued)

representation.[13] As the number 2 is represented by the character `3` in Base58 encoding (see Table 3.1), all the P2SH mainnet addresses are 34-character addresses beginning with the character `3`. This makes it easy to distinguish them from P2PKH mainnet addresses which all begin with the character `1`.

While bare m-of-n multisig scripts with n greater than 3 are non-standard, values of n upto 15 can be used in a P2SH m-of-n multisig script. The upper limit of 15 arises from the 520-byte cap on the size of an item which can be pushed onto the stack. During P2SH response script execution, the redeem script is pushed onto the stack limiting its maximum size to 520 bytes. The operators in a m-of-n multisig redeem script occupy 3 bytes leaving 517 bytes for the public keys. Each compressed public key occupies 34 bytes in the script where 33 bytes are needed for the key itelf and 1 byte is needed for the data push operator `0x21` which pushes the 33 bytes. As $15 \times 34 = 510$, a maximum of 15 public keys can be specified in the redeem script.

Null Data

The null data challenge script is a method to store small amounts of data (upto 80 bytes) on the blockchain. It has a `scriptPubkey` of the form

```
OP_RETURN <Data>
```

where `<Data>` can contain at most 80 bytes of arbitrary data. The `scriptPubkey` field itself has a maximum size of 83 bytes with the `OP_RETURN` operator occupying 1 byte, the length of the data occupying at most 2 bytes,[14] and the data occupying upto 80 bytes.

The `OP_RETURN` operator causes script execution to terminate immediately irrespective of the state of the stack. So there exists no response script which can provide a valid response to this challenge script. For this reason, null data outputs are unspendable and any bitcoins locked by a null data challenge script will be lost forever. Outputs containing null data challenge scripts are not added to the set of UTXOs even if they have some bitcoins associated with them.

The sole reason for including the null data challenge script in the list of standard scripts is to provide a means to securely store data on the blockchain. Once the block containing the null data output receives a few dozen confirmations, it becomes computationally infeasible to change the data recorded in the output. This property can be exploited to build timestamping applications where the hash of some document can be recorded on the blockchain to prove the existence of the document prior to some point in time.

[13] The base 58 representation of a positive integer N is the sequence of digits $a_k a_{k-1} \cdots a_0$ where $0 \leq a_i \leq 5$ and $N = \sum_{i=0}^{k} a_i 58^i$.

[14] The length of the data is encoded using the data push operators given in Table 5.2. Data lengths upto 75 bytes need only one byte to encode while data lengths from 76 to 80 need two bytes.

Signature Hash Type	Value
SIGHASH_ALL	0x00000001
SIGHASH_NONE	0x00000002
SIGHASH_SINGLE	0x00000003
SIGHASH_ANYONECANPAY	0x00000080

Table 5.4: Base signature hash types and their values

5.6 Pre-SegWit Signature Generation

When a regular transaction input unlocks a UTXO, the ECDSA signatures in the response script serve a dual purpose. In addition to proving ownership of the relevant private keys, they prevent the transaction from being tampered with before its inclusion in the blockchain. Recall that the ECDSA takes two inputs: a private key and a message digest. The message digest is a hash of the message being signed by the ECDSA. When a new regular transaction is created, the parts of the transaction which need to be protected from modification are included in the message used to generate the signature. If the message is modified, the message digest is also modified by the second preimage resistance of the hash function. The signature created using the unmodified message will no longer be valid for the modified message. When a regular transaction is broadcast on the Bitcoin network, nodes will not relay it if the signatures are invalid. The difference between the signature generation schemes for pre-SegWit and SegWit transactions lies in the construction of the message which is signed. In this section, we describe the pre-SegWit signature generation scheme and discuss the SegWit signature generation scheme in Section 5.10.

While the option of signing the entire transaction is available, the Bitcoin protocol has options where only some parts of the transaction are signed. The latter options allow the signer to intentionally allow the modification of the unsigned parts of the transaction. The available options are specified by the least significant byte of a 4-byte field called the *signature hash type*. The term signature hash is a synonym for the term message digest. The base signature hash types and their values are shown in Table 5.4. These hash types are applicable for both pre-SegWit and SegWit transactions.

SIGHASH_ALL

The SIGHASH_ALL hash type is the default option where all the inputs and outputs in the transaction are signed. Consider the regular transaction shown in Figure 5.15. The number 0x02 which appears once before the inputs and once before the outputs indicates there are two inputs and two outputs in the transaction. Input 0 unlocks a previous output which is at index n0 of a transaction with TXID hash0. Let prevScriptPubkey0 denote the scriptPubkey

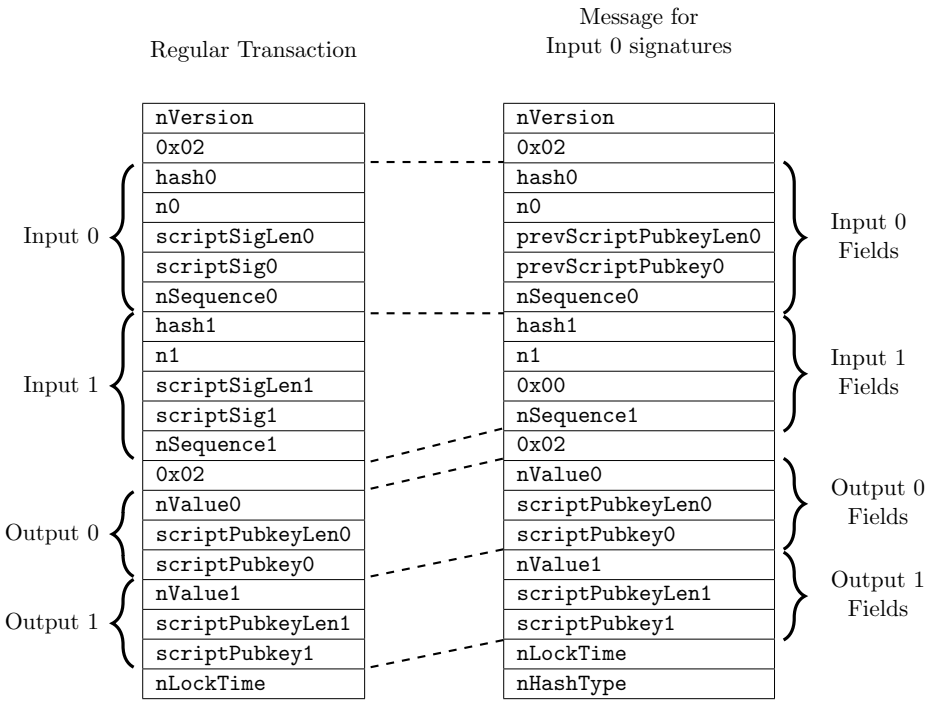


Figure 5.15: Message used to generate signature for the first input using the `SIGHASH_ALL` hash type

field of this output and let `prevScriptPubkeyLen0` denote its length. If the challenge script specified by `prevScriptPubkey0` requires a signature to be present in `scriptSig0` with signature hash type `SIGHASH_ALL`, then the message to be signed is shown on the right in Figure 5.15. This message is obtained according to the following rules:

- The `nVersion` field and the number of inputs are included without modification.
- The `scriptSig0` and `scriptSigLen0` fields in Input 0 are replaced with `prevScriptPubkey0` and `prevScriptPubkeyLen0` respectively.[15] The other fields in Input 0 are included without modification.
- The `scriptSig1` and `scriptSigLen1` fields in Input 1 are replaced with a single zero byte and the remaining fields are included without modification.

[15] If `prevScriptPubkey0` contains the `OP_CODESEPARATOR` operator, the Bitcoin Core client modifies `prevScriptPubkey0` before including it in the message. The `OP_CODESEPARATOR` operator is an artifact of a previous implementation of the script execution algorithm in the Bitcoin Core client. It is not used in any of the present standard scripts. To keep the exposition simple, we assume that `prevScriptPubkey0` does not contain it.

- The number of outputs, the fields in the outputs, and the `nLockTime` field are included without modification.
- The 4-byte signature hash type is appended at the end as shown by the `nHashType` field. This step is not unique to the `SIGHASH_ALL` signature hash type and is done for all the other hash types as well.

The message digest is calculated as the double SHA-256 hash of this message and signed with the private key to generate the signature. The signature will appear in the `scriptSig0` field of Input 0. As the signature is not known at the time of message digest calculation, the `scriptSig0` field cannot be included in the message. If any of the fields included in the message are modified after the signature generation, the signature will become invalid. This property has the following semantic consequences:

- Since `hash0` and `n0` are included in the message, the UTXO being unlocked by Input 0 cannot be changed. This is because `hash0` contains the TXID of the transaction containing the UTXO and `n0` contains the index of the UTXO in the list of outputs in that transaction. Specifying the UTXO location in the message prevents the signature generated to unlock the UTXO from being used again to unlock another UTXO locked by the same challenge script (for example, two UTXOs may have the same P2PKH challenge script).[16]
- As `hash1` and `n1` are included in the message, the UTXO which will be unlocked by Input 1 cannot be changed. This is useful in scenarios when the response scripts in Input 0 and Input 1 are provided by two different entities. The entity generating the signatures for Input 0 can be sure that the amount of bitcoins contributed by Input 1 will not change as the corresponding UTXO cannot change.
- As all the output fields are included in the message, the intended recipients (specified by the challenge scripts) of the bitcoins unlocked by the transaction inputs and the amounts being sent them cannot be changed.
- As the `nSequence0`, `nSequence1`, and `nLockTime` fields are included in the message, the absolute and relative lock time semantics of the transaction cannot be changed.
- As the signature hash type is included as part of the `nHashType`, the procedure used to generate the signature cannot be changed.

[16] Including the `prevScriptPubkey0` and `prevScriptPubkeyLen0` fields from the UTXO in the message with the intent of protecting them is technically unnecessary as the computation of the TXID `hash0` includes these fields in the double SHA-256 hash of the transaction. This inclusion was done in the original implementation of the Bitcoin Core client and remains as it will require a hard fork change to remove it.

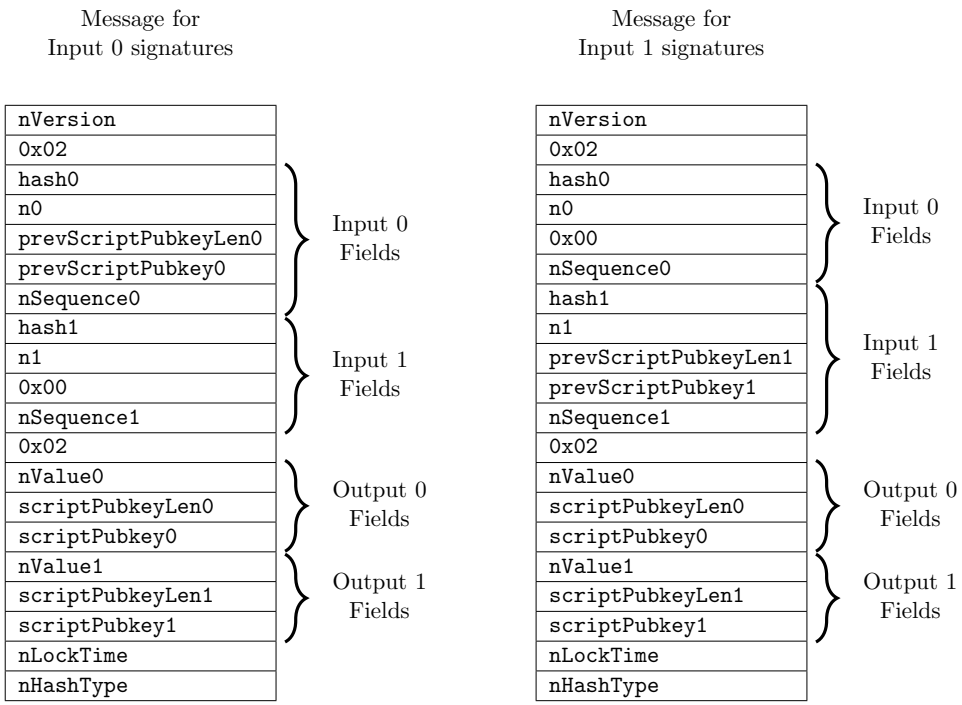


Figure 5.16: Message used to generate signature for the second input using the `SIGHASH_ALL` hash type

The message used to generate the signature for Input 1 of the transaction from Figure 5.15 using the `SIGHASH_ALL` hash type is shown in Figure 5.16. We have repeated the message used for the Input 0 signature in this figure for easy comparison between the two messages. The two messages differ only in which input fields are included. The message for Input 1 signatures replaces the `scriptSig0` and `scriptSigLen0` from Input 0 with a zero byte. This is done even if the `scriptSig0` field is already known to allow the signatures for the two inputs to be generated in any order. The `prevScriptPubkey1` and `prevScriptPubkeyLen1` be the challenge script and its length from the UTXO being unlocked by Input 1. These fields replace the `scriptSig1` and `scriptSigLen1` fields in Input 1.

While we used a transaction with two inputs and two outputs to illustrate the message generation for the `SIGHASH_ALL` hash type, the procedure for transactions with arbitrary number of inputs and outputs is similar. All the outputs in the transaction are included in the message. When generating the message for a particular input, the `scriptSig` and `scriptSigLen` fields in that input are replaced with the `scriptPubkey` and `scriptPubkeyLen` fields from the UTXO being unlocked. The `scriptSig` and `scriptSigLen` fields from all the other inputs are excluded.

As discussed in Section 5.5, for all signature hash types the ECDSA sig-

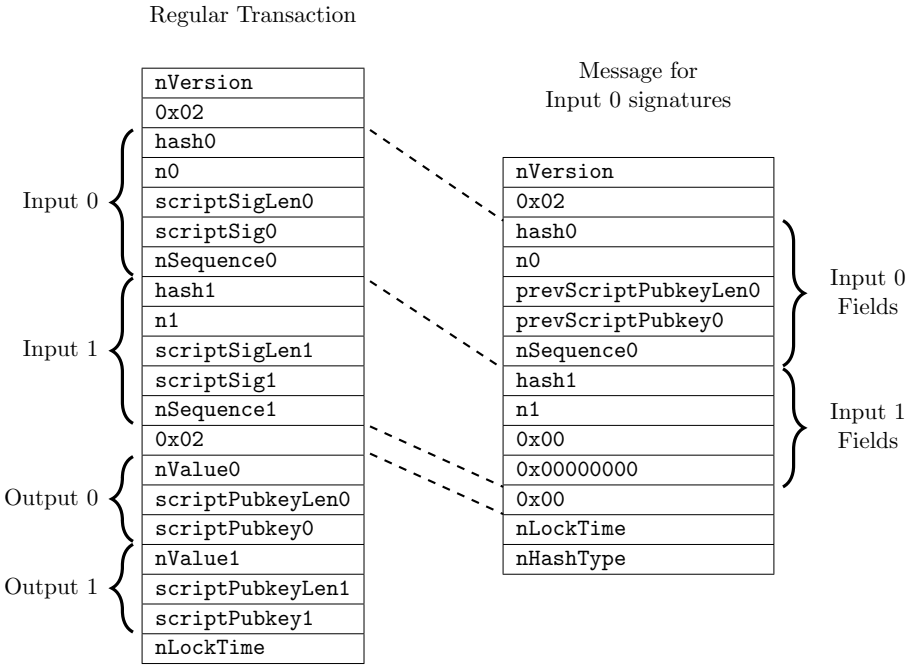


Figure 5.17: Message used to generate signature for the first input using the `SIGHASH_NONE` hash type

nature generated using the message digest and private key is encoded using DER encoding. The least significant byte of the signature hash type is appended to this DER encoded signature to indicate its type. This byte is used by the operators `OP_CHECKSIG` and `OP_CHECKMULTISIG` to generate the correct message from the transaction for signature verification.

`SIGHASH_NONE`

When the `SIGHASH_NONE` signature hash type is used, none of the outputs in the transaction are included in the message being signed. Figure 5.17 shows the message used to generate the signatures for the first input of the transaction from Figure 5.15. The fields related the inputs of the transaction are obtained by the same procedure used in the `SIGHASH_ALL` case with one exception. The `nSequence1` field from Input 1 is set to zero (`0x00000000`) in the message. This allows the `nSequence1` field to be modified before the signatures in Input 1 are generated.

The field indicating number of outputs is set to zero and none of the fields from the outputs of the transactions are included in the message. This may seem insecure because the outputs can be modified without invalidating the signature. If a regular transaction in which all the inputs have signatures with `SIGHASH_NONE` hash type is broadcast on the network, the miner can replace

the addresses receiving the payment with its own address and include the transaction in the blockchain. This can be avoided by having at least one of the transaction inputs have signatures with the `SIGHASH_ALL` hash type.

The utility of the `SIGHASH_NONE` hash type is that it enables entities which trust each other to construct transactions where the receiver of the payment is not known beforehand. For example, suppose Alice and Bob want to purchase a rare book by pooling their bitcoin funds. They will create a transaction containing two inputs where each of them will provided signatures for one of the inputs. But the book is not currently available and requires them to search for a seller who has it. Bob offers to search for the book. If Alice trusts Bob, she can unlock the UTXO containing her contribution to the cost of the book using signatures of `SIGHASH_NONE` hash type and give the transaction to Bob. The message used to generate Alice's signature will not contain the outputs. Hence the receiver of the payment is not fixed. When Bob finds a seller having the book, he can include the seller's Bitcoin address in the transaction outputs and unlock the UTXO containing his contribution using signatures of `SIGHASH_ALL` hash type. When this transaction is broadcast on the network, the outputs cannot be modified as they are protected by the signatures in Bob's input.

For transactions with arbitrary number of inputs and outputs, the message for generating the signatures for each input always excludes the outputs and sets the number of outputs to zero. Apart from setting the `nSequence` fields in all the other inputs to zero when generating the message for a particular input, the portion of the message related to the transaction inputs is generated as in the `SIGHASH_ALL` case.

`SIGHASH_SINGLE`

The `SIGHASH_SINGLE` signature hash type is used in situations when each entity unlocking a UTXO in a multi-input transaction wants to sign only one of the outputs. In the message for signatures in the input at index i, only the output at index i is included. The fields related to the inputs are included in the message by the same procedure used in the `SIGHASH_NONE` case.

Figure 5.18 shows the messages used to generate the signatures in each of the two inputs of the transaction from Figure 5.15. For the Input 0 message, the number of outputs is set to one (`0x01`) and only the fields from Output 0 are included in the message. For the Input 1 message, the number of outputs is set to two (`0x02`). A null output is included instead of Output 0. It consists of a 64-bit `nValue` field set to all ones (`0xFFFF FFFF FFFF FFFF`) followed by a single zero byte representing an empty `scriptPubkey` field. All the fields from Output 1 are included in the message. In general, the message for an input with index i includes the number of outputs set to $i + 1$, $i - 1$ null outputs, and the unmodified output with index i. The outputs with index greater than i are ignored.

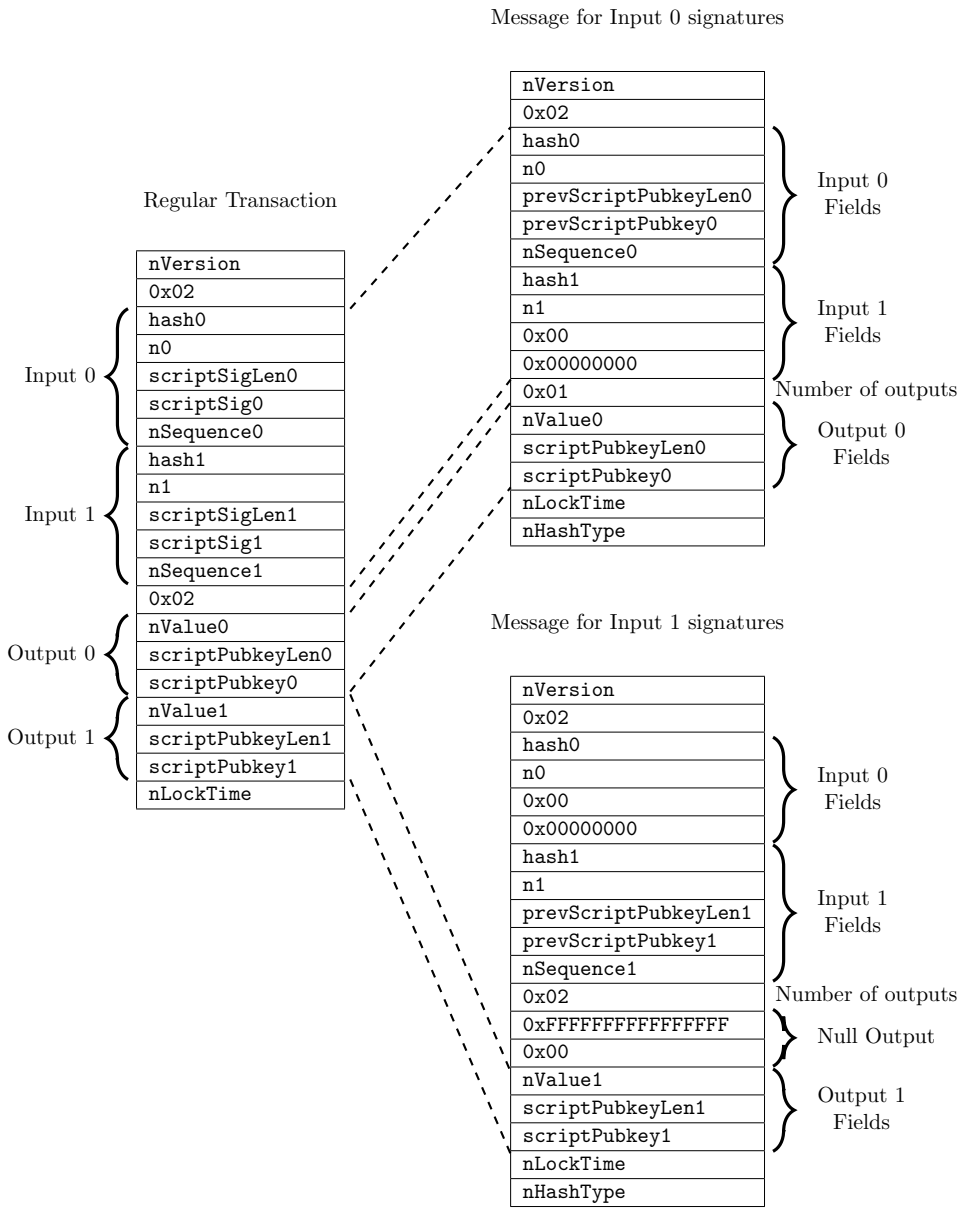


Figure 5.18: Messages used to generate signatures for each input using the `SIGHASH_SINGLE` hash type

If the number of outputs is less than the number of inputs, an input at index i may not have a corresponding output at index i. In this case, if the `SIGHASH_SINGLE` hash type is used to generate the signature for the input at index i, the Bitcoin Core client sets the message digest to a 256-bit string consisting of 255 zeros followed by a single one. This is a bug as it makes the signature independent of the message. A signature of the `SIGHASH_SINGLE`

hash type which was used unlock a UTXO can be reused to unlock other UTXOs locked by the same challenge script. This bug is due to an oversight in an early implementation of the Bitcoin Core client and has remained unfixed as it requires a soft fork to fix.

The `SIGHASH_SINGLE` hash type is useful in scenarios when multiple entities fund the different inputs in a transaction. Each entity wants to ensure that a certain amount of bitcoins from the inputs is paid to a receiver of their choice. But each entity does not care for how the remaining amount of bitcoins are spent.

`SIGHASH_ANYONECANPAY`

All previous three signature hash types include all the inputs of the transaction in the message being signed. The `SIGHASH_ANYONECANPAY` hash type specifies that the message used to generate signatures for a particular transaction input includes only that input. It is always used in conjunction with one of the three previous signature hash types. For example, the `SIGHASH_ANYONECANPAY|SIGHASH_ALL` hash type corresponds to the case when all the outputs are signed and only one of inputs is signed. This hash type is represented by the value `0x81` which is the bitwise OR of the `SIGHASH_ANYONECANPAY` and `SIGHASH_ALL` hash type values from Table 5.4. Once the signature of this hash type has been generated, the outputs in the transaction cannot be modified but the other inputs can be modified. Hence the name "anyone can pay".

Figure 5.19 shows the messages used to generate the signatures having hash type `SIGHASH_ANYONECANPAY|SIGHASH_ALL` in each of the two inputs of the transaction from Figure 5.15. For the Input 0 message, the number of outputs is set to one (`0x01`) and only the fields from Input 0 are included in the message. For the Input 1 message, the number of outputs is once again set to one and only the fields from Input 1 are included. In both cases, all the output fields are included in the message.

The `SIGHASH_ANYONECANPAY|SIGHASH_NONE` and `SIGHASH_ANYONECANPAY|SIGHASH_SINGLE` hash types have values `0x82` and `0x83`. The messages generated by these hash types include only one input at a time. The inclusion of the outputs follows the procedure described for `SIGHASH_NONE` and `SIGHASH_SINGLE` respectively.

To see the utility of the `SIGHASH_ANYONECANPAY` hash type, consider a crowdfunding scenario where the recipient of the funds and the amount of funds required are known. Some people want to participate in the crowdfunding by unlocking UTXOs containing their contribution. A transaction is created where each funder's contribution is represented by an input and the output contains the recipient's address. Without using the `SIGHASH_ANYONECANPAY` hash type, the number of funders and their UTXO details will need to be fixed before the signatures for each input can be generated. But with the `SIGHASH_ANYONECANPAY` hash type, the signatures for each input can be gen-

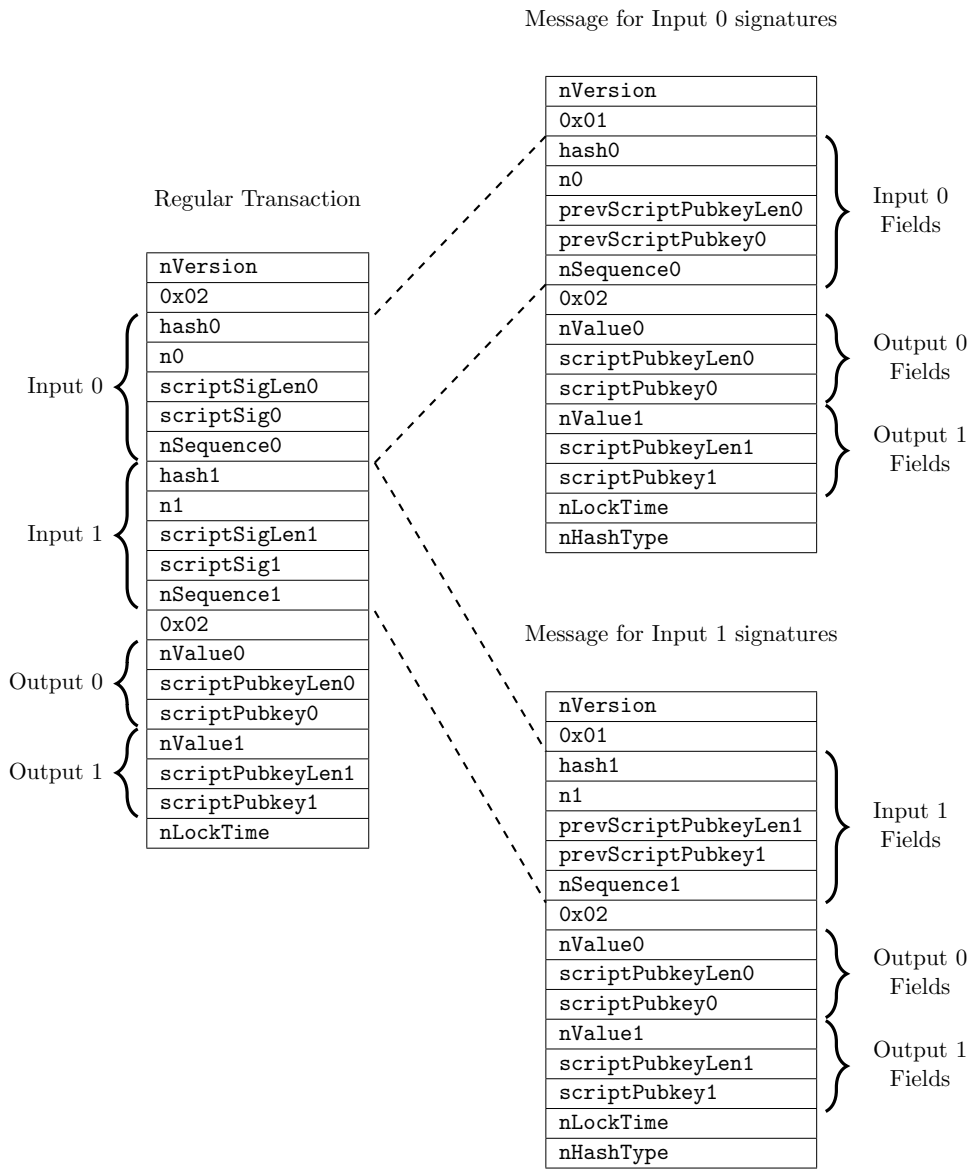


Figure 5.19: Messages used to generate signatures for each input using the `SIGHASH_ANYONECANPAY|SIGHASH_ALL` hash type

erated without knowing these details in advance. The transaction becomes valid once the sum of the bitcoin amounts unlocked by the inputs exceeds the amount in the output. Note that the excess amount will be transferred to the miner who includes this block on the blockchain as transaction fees.

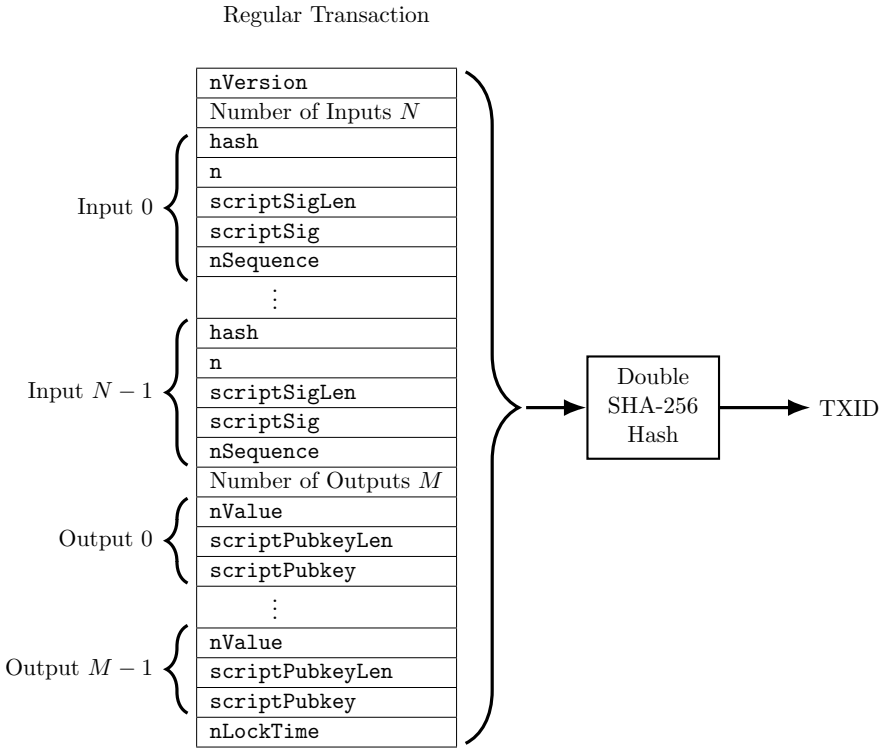


Figure 5.20: TXID calculation by double SHA-256 hash of transaction fields

5.7 Transaction Malleability

The TXID of a transaction is obtained by calculating the double SHA-256 hash of all the fields in it. Figure 5.20 shows the fields which are hashed for a regular transaction with N inputs and M outputs. Due to the collision resistance of the SHA-256 hash function, changing any of the fields in the transaction will result in a different TXID. Transaction malleability refers to the phenomenon where the TXID of a transaction can be modified without making a *functional change* to the transaction. Two transactions are identical from a functional viewpoint if the following conditions hold:

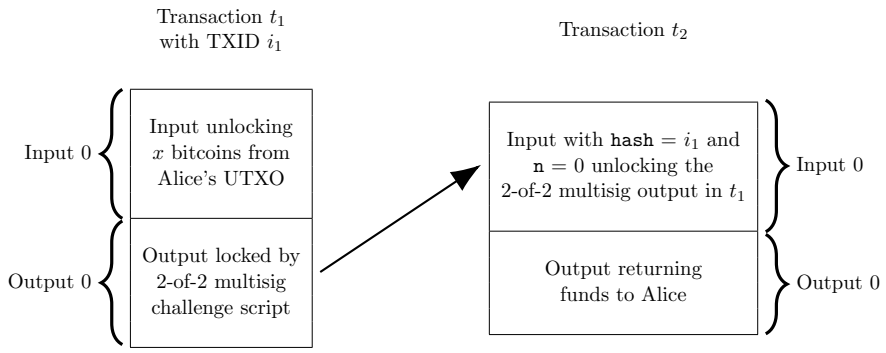
- They both specify the same previous UTXOs as the source of bitcoins. The `hash` and `n` fields in the inputs of both the transactions have to be identical.
- They both specify the same new UTXOs as destinations of the bitcoins being transferred. The `nValue` and `scriptPubkey` fields in the outputs of both transactions have to be identical.
- The lock time restrictions imposed by the `nSequence` and `nLockTime` fields in both the transactions are identical.

Note that the `scriptSig` fields are not required to be identical for two transactions to be functionally identical. The `scriptSig` field contains ECDSA signatures whose generation involves a random integer (see Section 2.5). Hence there are multiple valid signatures corresponding to the same message digest and private key. Each of these signatures will result in a different `scriptSig` field in an input and consequently a different TXID for the transaction. So an entity which knows *any one* of the private keys needed to generate a signature required in the response script can change the TXID by simply regenerating the signature with a different random integer.

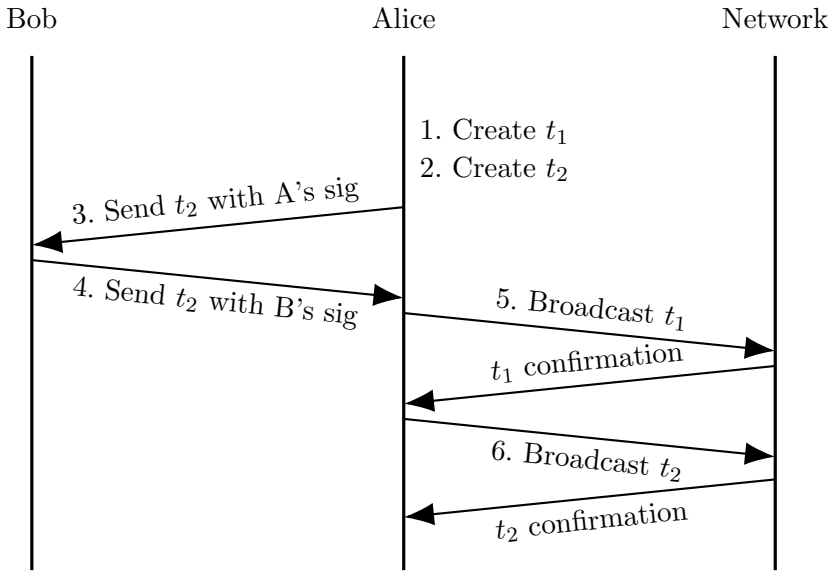
Transaction malleability can even be effected by entities which do not know any of the private keys required to generate a valid response script. Using notation from Section 2.5, let (r, s) be a valid `secp256k1` ECDSA signature for a message m. In Appendix A, we show that $(r, n-s)$ is also a valid signature for the message m where p is the 256-bit prime number given in equation (2.3). The integer s is not allowed to be zero in an ECDSA signature. Since n is an odd integer, $n - s \neq s \bmod n$ for all $s \neq 0$. Hence replacing the byte representation of (r, s) in a `scriptSig` field of a transaction with the byte representation of $(r, n - s)$ will modify the TXID without invalidating the signature. This change does not require knowledge of the private key which was used to generate (r, s).

Once a transaction receives a confirmation i.e. once it is included in a block on the blockchain, changing its TXID will change the `hashMerkleRoot` in the block header. With a high probability, this will cause the block hash to exceed the target threshold invalidating the block. As a result, transaction malleability does not pose problems in situations when the transaction outputs will be spent after the transaction is confirmed. However, there are protocols where unconfirmed transaction outputs are referenced by a spending transaction. Transaction malleability will cause such protocols to fail by breaking the dependency between the transactions.

To understand how transaction malleability causes protocols involving spending of unconfirmed transaction outputs to fail, consider the following situation. Suppose Alice is a professor who wants to teach her student Bob about Bitcoin transactions. Bob does not own any bitcoins. So Alice decides to transfer x bitcoins to Bob with the intention of getting them back. But Alice does not have enough trust in Bob's integrity or competence to send the bitcoins to a Bitcoin address which requires only Bob's signature to spend (for example, a P2PK or P2PKH address derived from Bob's private key). Bob may decide to cheat Alice by refusing to provide the signature required to send the bitcoins from his Bitcoin address back to Alice. Or he may make a mistake in constructing the refund transaction resulting in the bitcoins being sent to an address not owned by Alice. To prevent these undesirable situations, Alice executes a protocol which is illustrated in Figure 5.21. Figure 5.21(a) illustrates the transactions t_1 and t_2 Alice creates to initiate the protocol and Figure 5.21(b) illustrates the messages exchanged during the protocol. The



(a) Illustration of the transactions t_1 and t_2 used in Alice's protocol.



(b) Messages exchanged in Alice's protocol

Figure 5.21: Alice's protocol for transferring funds to a 2-of-2 multisig output and reclaiming them

protocol proceeds in the following manner:

1. Alice creates a transaction t_1 which transfers x bitcoins from a UTXO she owns to an output that is locked by a 2-of-2 multisig challenge script. The 2-of-2 multisig challenge script contains two public keys, one each owned by Alice and Bob. A valid response script to this challenge script will contain signatures from both Alice and Bob. But Alice does not broadcast t_1 on the network.

2. She then creates a refund transaction t_2 which spends the output in t_1 and transfers the bitcoins back to her Bitcoin address. Initially, t_2's

input contains the TXID i_1 of t_1 and the index of the output containing the 2-of-2 multisig challenge script. But the response script in t_2's input has no signatures.

3. Alice includes her signature in the response script in t_2 and sends it to Bob. She asks him to add his signature in the response script and return t_2 to her. If Bob refuses to do this, Alice can stop the protocol. She has not lost any funds as t_1 has not been recorded on the blockchain.

4. If Bob includes his signature in t_2 and returns it to Alice, both Alice and Bob have fully signed the refund transaction which spends the 2-of-2 multisig output in t_1 to refund Alice.

5. Alice now broadcasts t_1 on the network.

6. Once t_1 is included in a block, either Alice or Bob can broadcast t_2 on the network. After t_2 is included in a block, the x bitcoins minus the transaction fees have returned to Alice.

Even though Alice creates t_1 in step 1, she does not broadcast it on the network until Bob sends her the refund transaction t_2 with his signature included in it. The dependency between t_1 and t_2 is created using the TXID i_1 of t_1.

Transaction malleability can be used to cause Alice's protocol to fail by breaking the dependency between t_1 and t_2. When Alice broadcasts t_1 in step 5, Bob or any node on the network can replace Alice's signature (r, s) in t_1 with the valid signature $(r, n - s)$ and rebroadcast the modified t_1 on the network. Let t_1' denote the modified t_1 that has a different TXID i_1'. Since both t_1 and t_1' spend the same UTXO owned by Alice, only one of them will be included in a block. If t_1' gets included in a block, then the refund transaction t_2 is invalid as its input contains the TXID i_1 of t_1. Now Alice will have to request Bob to sign a new version of t_2 in order to get her bitcoins back. If Bob refuses to cooperate, then she cannot get her bitcoins back. While Bob cannot spend the bitcoins as this requires Alice's signature, he can inconvenience Alice by making the funds unspendable.

The reason transaction malleability is possible is because all the signature hash types exclude the `scriptSig` field from the message digest used to create signatures in a transaction. But this field is included in the TXID calculation. Replacing the signatures in the `scriptSig` field with new valid signatures changes the TXID without invalidating the transaction. SegWit solves the problem of transaction malleability by defining two new script templates which move the signature data out of the `scriptSig` field and into a separate structure called the *witness*. This witness structure is not included in the TXID calculation. In the Bitcoin Core client, the witness structure is stored in a field called `scriptWitness`.

5.8 SegWit Standard Scripts

SegWit adds two new challenge script templates to the set of standard script templates: Pay to Witness Public Key Hash (P2WPKH) and Pay to Witness Script Hash (P2WSH). Both of these templates can be embedded inside a P2SH template and the resulting templates are called P2SH-P2WPKH and P2SH-P2WSH respectively.

The first byte in SegWit challenge scripts is used to indicate the *script version*. P2WPKH and P2WSH are version 0 scripts which is indicated by a zero byte. SegWit allows for future definition of script versions 1 through 16.

Pay to Witness Public Key Hash (P2WPKH)

The P2WPKH script template is the SegWit analog of the P2PKH script template. The P2WPKH challenge script template has a 22-byte `scriptPubkey` field of the form

```
OP_0 0x14 <PubKeyHash>
```

where the `OP_0` operator indicates that it is a version 0 script and `<PubKeyHash>` is the 20-byte SHA-256 + RIPEMD-160 hash of a compressed public key. The `0x14` operator pushes the 20-byte hash onto the stack.

Recall that the script execution for pre-SegWit scripts consists of the execution of the script in the `scriptSig` field on an empty stack followed by the execution of the script in the `scriptPubkey` field. If we consider the pre-SegWit script execution procedure for the P2WPKH challenge script, then an empty `scriptSig` field is a valid response script. An empty response script does nothing but the subsequent execution of the P2WPKH consists of a push of an empty byte array (due to `OP_0`) followed by the push of the 20-byte `<PubKeyHash>` field. Thus the `<PubKeyHash>` is the top stack element when the script execution completes. Since `<PubKeyHash>` is extremely unlikely to be the all zeros bytestring, it evaluates to `True` and the script execution succeeds. This is illustrated in Figure 5.22. The implication is that a transaction output which is locked by a P2WPKH challenge script appears like an *anyone-can-spend output* to nodes which have not upgraded to SegWit-capable clients. On the other hand, SegWit-capable clients perform a different script execution procedure for SegWit scripts. To unlock the output locked by a P2WPKH challenge script, both the public key whose hash is equal to `<PubKeyHash>` and a valid signature created using the corresponding private key are needed. As long as nodes which control a majority of the network hashrate run SegWit-capable clients, transactions which attempt to spend P2WPKH outputs without providing the required signatures will not be included in the blockchain.

A valid response to a P2WPKH challenge script consists of an empty `scriptSig` field and a `scriptWitness` field which has a valid signature and a compressed public key whose SHA-256 + RIPEMD-160 hash is equal to

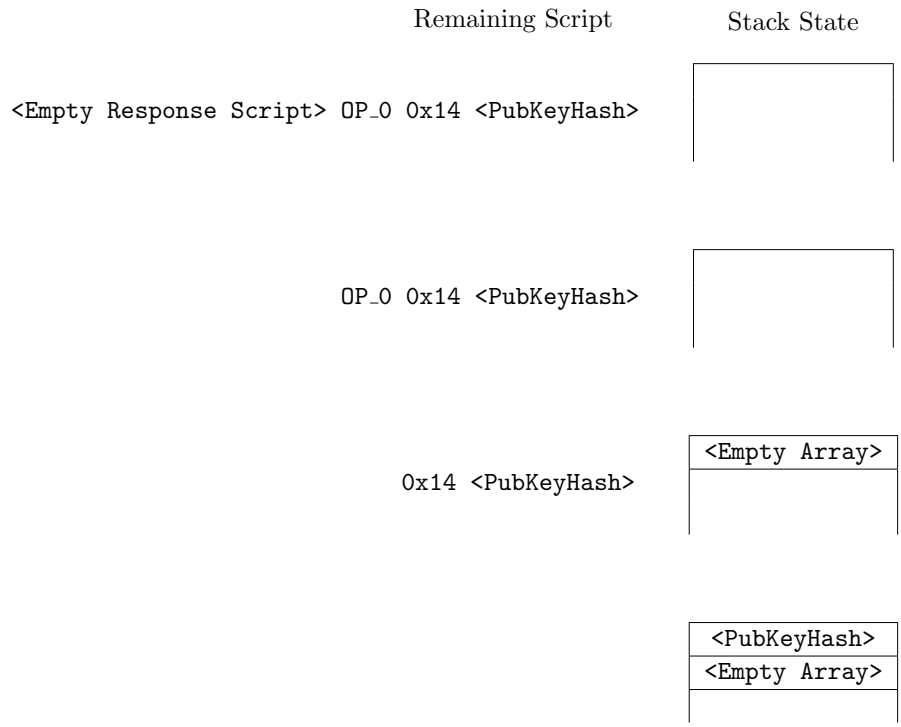


Figure 5.22: Stack state during the execution of an empty response script followed by a P2WPKH challenge script

<PubKeyHash>, i.e.

```
scriptSig:     (empty),
scriptWitness: <Signature> <Public Key>.
```

The location of the scriptWitness field in the spending transaction is described in Section 5.9. To verify that the response to a P2WPKH challenge script is valid, a SegWit-capable client first constructs the following P2PKH challenge script using the <PubKeyHash> field in the challenge script.

```
OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG.
```

The fields in the scriptWitness are pushed onto an empty stack and the P2PKH challenge script is executed on this stack. Note that the fields in the scriptWitness are identical to the scriptSig fields of the P2PKH response script described in Section 5.5. The script execution is as shown in Figure 5.11.

Suppose a miner mines a new valid block which has a transaction with an input which spends P2WPKH outputs. When this block is broadcast on the network, nodes running SegWit-capable clients receive both the scriptSig

and `scriptWitness` fields. But nodes running pre-SegWit clients do not receive the `scriptWitness` fields. They see only the empty `scriptSig` fields which are valid responses as per the pre-SegWit script execution procedure. Hence the block is considered valid by the nodes running pre-SegWit clients.

While each transaction input which unlocks an output locked by a SegWit challenge script has a `scriptWitness` field associated with it, this field is not included in the input data structure to maintain backward compatibility with pre-SegWit clients. The signature data which was part of the `scriptSig` field in pre-SegWit regular transactions is moved to the `scriptWitness` field. In cryptography, digital signatures are considered witnesses which prove that the signer knows the private key corresponding to a public key. The phrase "segregated witness" is motivated by the separation of the signature data from the inputs.

The P2WPKH challenge script can be embedded inside a P2SH script template resulting in a P2SH-P2WPKH challenge script. The `scriptPubkey` field has a P2SH challenge script of the form

```
OP_HASH160 <RedeemScriptHash> OP_EQUAL
```

where `RedeemScriptHash` is the 20-byte SHA-256 + RIPEMD-160 hash of the P2WPKH challenge script. The `scriptSig` and `scriptWitness` fields are given by

$$\texttt{scriptSig:}\quad \texttt{0x16}\ \underbrace{\texttt{OP\_0 0x14 <PubKeyHash>}}_{\text{Redeem Script}},$$

$$\texttt{scriptWitness:}\quad \underbrace{\texttt{<Signature> <Public Key>}}_{\text{Response to Redeem Script}}.$$

The `0x16` operator in the `scriptSig` field pushes the 22-byte P2WPKH challenge script (which is the P2SH redem script) onto the stack. The `scriptWitness` field once again contains a signature and a compressed public key. While the response to the redeem script was present in the `scriptSig` field in pre-SegWit P2SH scripts, it is located in the `scriptWitness` field in the P2SH-P2WPKH script. The advantage of a P2SH-P2WPKH script template over the P2WPKH script template is that an entity requesting bitcoin payment to an output locked by the former can simply share the P2SH address generated from the `RedeemScriptHash`.

When pre-SegWit clients encounter a transaction input which unlocks a P2SH-P2WPKH output, they will first verify that the SHA-256 + RIPEMD-160 hash `RedeemScriptHashCalc` of the redeem script in the `scriptSig` field matches the `RedeemScriptHash` given in the `scriptPubkey` of the output. When they execute the redeem script, the `OP_0` operator pushes an empty byte array and the `0x14` operator pushes the 20-byte `PubKeyHash` onto the stack making it the top stack element. As `PubKeyHash` is extremely unlikely

to be the all zeros bytestring, it evaluates to `True` and the script execution succeeds. This is illustrated in Figure 5.23. On the other hand, SegWit-capable clients will insert the `PubKeyHash` in the redeem script into a P2PKH challenge script as before and use the contents of the `scriptWitness` field to execute it.

Pay to Witness Script Hash (P2WSH)

The P2WSH script template is the SegWit analog of the P2SH script template. The P2WSH challenge script template has a 34-byte `scriptPubkey` field of the form

`OP_0 0x20 <RedeemScriptHash>`

where `RedeemScriptHash` is the 32-byte SHA-256 hash of a redeem script contained in the witness field. The `0x20` operator pushes this 32-byte hash onto the stack. As in the P2WPKH case, the `OP_0` operator indicates that this `scriptPubkey` contains a version 0 SegWit script.

A valid response to a P2WPKH challenge script consists of an empty `scriptSig` field and a `scriptWitness` field which has a valid response to the redeem script and a byte array containing the redeem script itself, i.e.

`scriptSig:` (empty),

`scriptWitness:` `<Response To Redeem Script> <Redeem Script Byte Array>`.

In general, the `scriptWitness` contains a list of stack items. The `<Redeem Script Byte Array>` is the entire redeem script represented as a single stack item.

As in the P2WPKH case, a pre-SegWit client will consider the empty `scriptSig` field as a valid response to the P2WSH challenge script. The response and challenge script execution will be as illustrated in Figure 5.22 with one difference. The top stack element at the end of script execution will be a 32-byte hash instead of the 20-byte hash in the P2WPKH case.

On the other hand, a SegWit-capable client will first verify that the SHA-256 hash of the redeem script byte array in the `scriptWitness` field is equal to `RedeemScriptHash`. If it is not equal, the script execution fails. Otherwise, the data in the `scriptWitness` which comprises the response to the redeem script is pushed onto the stack and the redeem script is executed against this stack. If the redeem script execution succeeds, the response is considered valid. This procedure is as illustrated in Figure 5.13 with the `OP_HASH160` operator replaced by the `OP_SHA256` operator which replaces the top stack element with its SHA-256 hash. While the P2SH script validation procedure takes the redeem script and the response to it from the `scriptSig` field, the P2WSH validation takes these fields from the `scriptWitness` field.

The P2WPKH challenge script can also be embedded inside a P2SH script template to take advantage of P2SH addresses. The challenge script itself is the redeem script for the P2SH script. To avoid confusion between the two

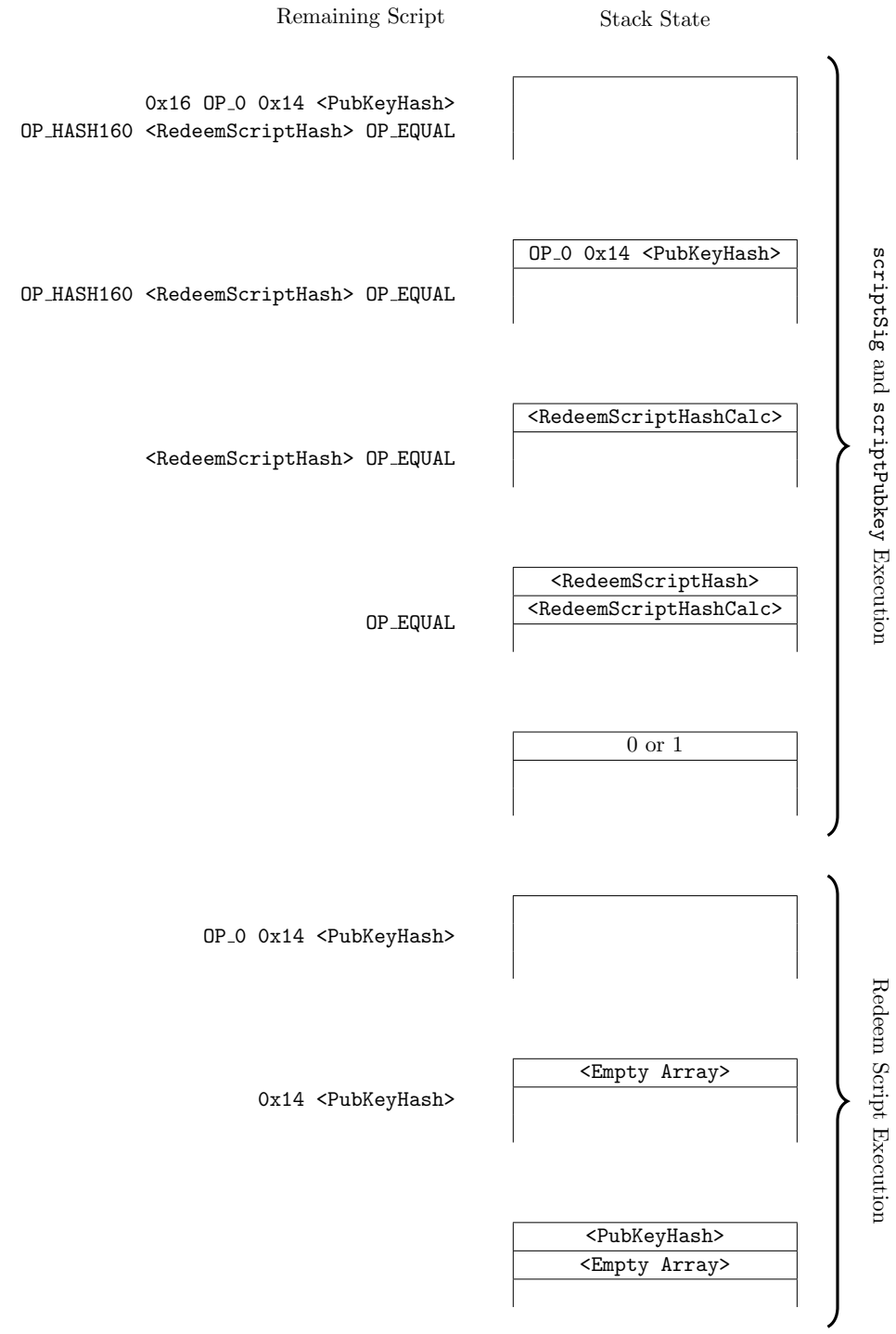


Figure 5.23: Stack state during P2SH-P2WPKH script execution by a pre-SegWit client

redeem scripts, we will call the redeem script in the `scriptSig` field the *P2SH redeem script* and the redeem script in the `scriptWitness` field the *P2WSH redeem script*. The `scriptPubkey` field has a P2SH challenge script of the form

`OP_HASH160 <P2SH RedeemScriptHash> OP_EQUAL`

where `P2SH RedeemScriptHash` is the 20-byte SHA-256 + RIPEMD-160 hash of the P2WSH challenge script (which is the P2SH redeem script). The `scriptSig` and `scriptWitness` fields are given by

`scriptSig:` `0x22` $\underbrace{\texttt{OP\_0 0x20 <P2WSH RedeemScriptHash>}}_{\text{P2SH Redeem Script}}$,

`scriptWitness:` `<Response To P2WSH Redeem Script>`
`<P2WSH Redeem Script Byte Array>`.

The `0x22` operator in the `scriptSig` field pushes the 34-byte P2WSH challenge script onto the stack. The `scriptWitness` field once again contains the response to the P2WSH redeem script followed by the script itself.

As in the P2SH-P2WPKH case, pre-SegWit clients validate inputs spending P2SH-P2WSH outputs based on the equality of the P2SH redeem script hash and the hash given in the `scriptPubkey` field. SegWit-capable clients perform the additional step of executing the P2WSH redeem script in the `scriptWitness` field.

5.9 SegWit Regular and Coinbase Transactions

A transaction output which is locked by a SegWit challenge script (P2WPKH, P2WSH, P2SH-P2WPKH, or P2SH-P2WSH) is called a *SegWit output*. If a transaction input unlocks a SegWit output, it is called a *SegWit input*. A regular transaction can have any combination of SegWit and pre-SegWit inputs and outputs. If a regular transaction has at least one SegWit input, it is called a *SegWit regular transaction*. The implication is that at least one of the inputs in a SegWit regular transaction has a non-trivial witness structure.

A SegWit regular transaction has two serialization formats (byte representations). The first serialization excludes the witness structures and looks exactly like the serialization of a pre-SegWit regular transaction. The double SHA-256 hash of this serialization yields the TXID of the transaction as illustrated in Figure 5.24. When a SegWit regular transaction is sent to a pre-SegWit client, this serialization of the transaction is used.

The second serialization of a SegWit regular transaction differs from the first in the following manner:

1. The `nVersion` field is followed by a 1-byte marker which contains `0x00`.
2. The marker byte is followed by a flag byte which contains `0x01`.

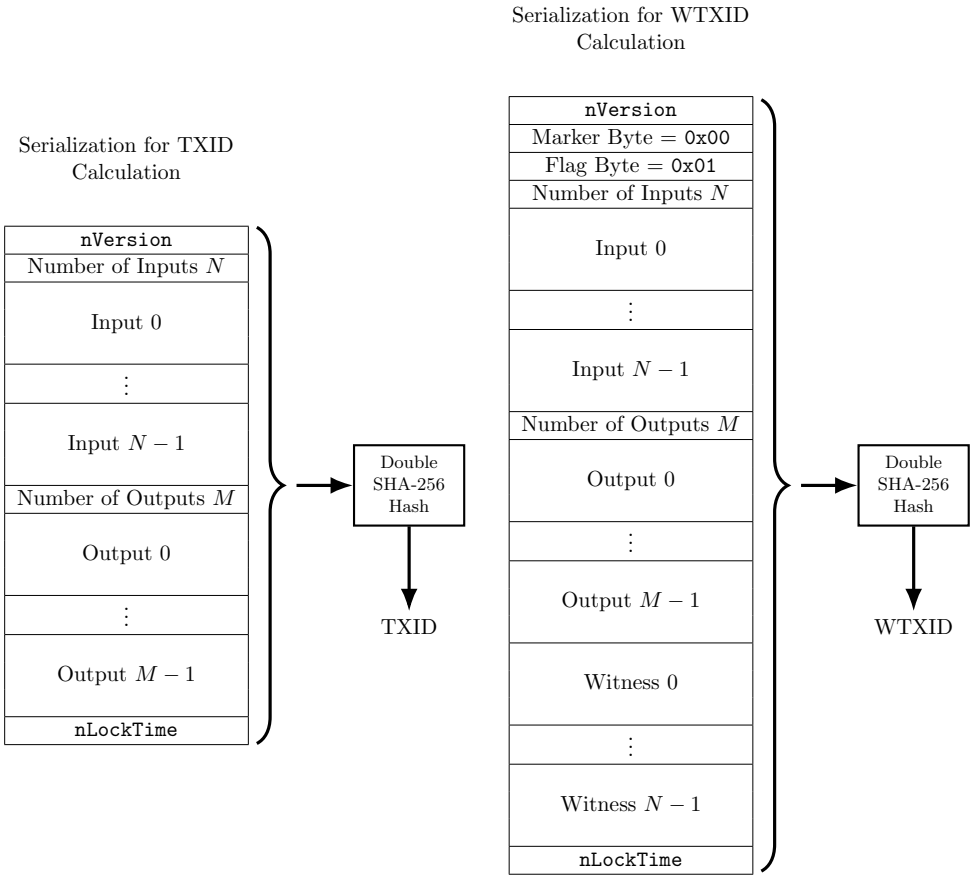


Figure 5.24: SegWit regular transaction serialization formats

3. If the transaction has N inputs, there are N witness structures between the outputs and the `nLockTime` field. The ith witness structure corresponds to the ith input. If an input is not a SegWit input, its corresponding witness structure is just a single byte containing `0x00`. The witness structures corresponding to SegWit inputs consist of a list of stack items. The first field in a witness structure is a VarInt which specifies the number of stack items in the witness structure as illustrated in Figure 5.25. This is followed by the stack items themselves where each stack item is preceded by a VarInt specifying its length in bytes.

The double SHA-256 hash of the second serialization is called the *witness transaction identifier (WTXID)*. If *all the inputs* in a transaction are SegWit inputs, the TXID is not malleable as its calculation does not involve any signatures which will be present in the witness structures. Even if one of the inputs in a transaction is a non-SegWit input, then the TXID of the transaction is malleable. The WTXID of transaction will be malleable as

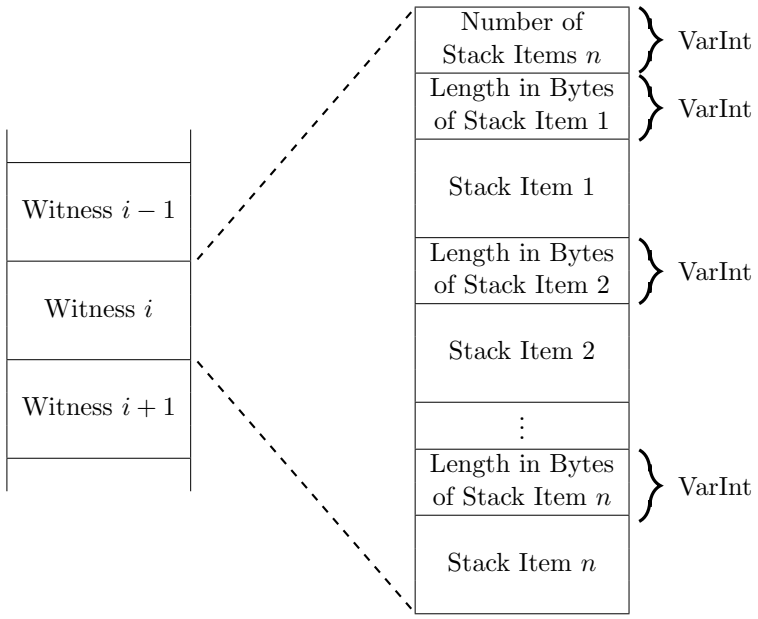


Figure 5.25: Witness structure format

its calculation involves the witness structures. As UTXOs are identified by the TXID of the transaction they are included in and the output index, the malleability of WTXIDs is not an issue.

Why is the WTXID calculated and how is it used? Recall that the root hash of the Merkle tree formed using the TXIDs is included in the `hashMerkleRoot` field of the block header (see Section 4.2). The motivation was that any tampering of the transaction data in a valid block, i.e. a block whose block hash falls below the target threshold, will cause the block hash to exceed the threshold invalidating it. With the exclusion of the signature data from the TXID calculation for SegWit inputs, there was a need to protect the integrity of this data from tampering. While invalid signatures can be detected by performing ECDSA signature validation, this is computationally expensive compared to verifying that the block hash falls below the target threshold. To make the block hash dependent on the WTXIDs, a new Merkle tree is constructed using the WTXIDs of all the transactions in a block as leaves using the following rules:

- The WTXID of the coinbase transaction is fixed to be the all zeros bytestring `0x0000...0000`.
- For non-SegWit transactions, the TXID is taken instead of its WTXID.

This is illustrated in Figure 5.26 where the Merkle tree corresponding to a block with only four transactions is shown. In the figure, t_0 is the coin-

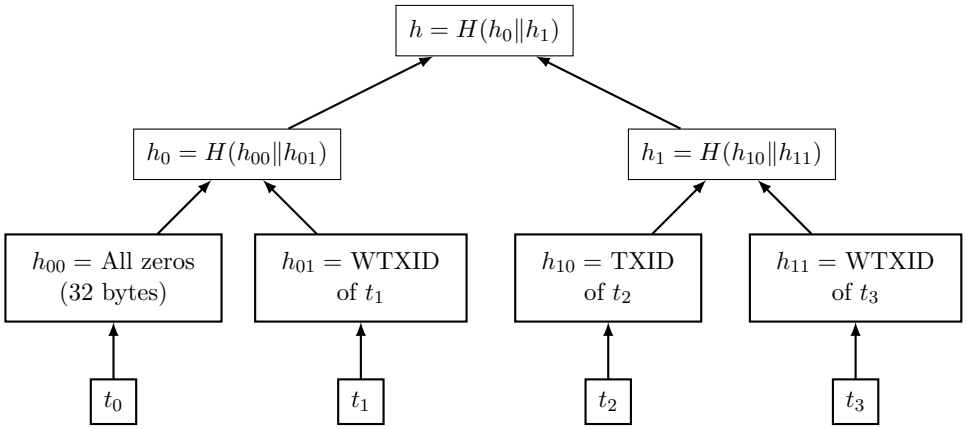


Figure 5.26: WTXID Merkle tree for a block with four transactions

base transaction, t_1 and t_3 are SegWit regular transactions, and t_2 is a non-SegWit regular transaction. The function $H(\cdot)$ represents the double SHA-256 function and $\|$ represents the concatenation operator. The root hash of the WTXID Merkle tree is called the *witness root hash*.

The double SHA-256 hash of the concatenation of the witness root hash and a 32-byte *witness reserved value* is then calculated and stored in the `scriptPubkey` field of a coinbase output. The witness reserved value is obtained from the coinbase transaction input's witness structure. This reserved value does not have any functionality associated with it for now and has been included to accommodate future changes. The coinbase output storing the witness root hash is a null data output with `scriptPubkey` of the form

$$\texttt{OP\_RETURN 0x24}\ \underbrace{\texttt{0xAA21A9ED}}_{\text{Commitment header}}\ \underbrace{H(\text{Witness root hash}\|\text{Witness reserved value})}_{\text{Commitment hash (32 bytes)}}$$

where the `0x24` operator indicates that the following 36 bytes are of interest. The 4-byte field containing `0xAA21A9ED` is the fixed commitment header for SegWit witness commitments. Figure 5.27 illustrates a coinbase transaction for a block containing SegWit transactions. There are two outputs in the transaction: the first output is a regular P2PKH output used by the miner to send the block reward to a P2PKH address owned by him and the second output is a null data output containing the witness commitment hash. The second output has a `nValue` field equal to zero as null data outputs are unspendable. The `scriptPubkeyLen` field is set to `0x26` indicating that the `scriptPubkey` is 38 bytes long. The `scriptPubkey` field contains the null data challenge script containing the witness commitment hash. Even though the dummy input in the coinbase transaction is not a SegWit input, it has a witness structure containing a single stack item consisting of the 32-byte witness reserved value.

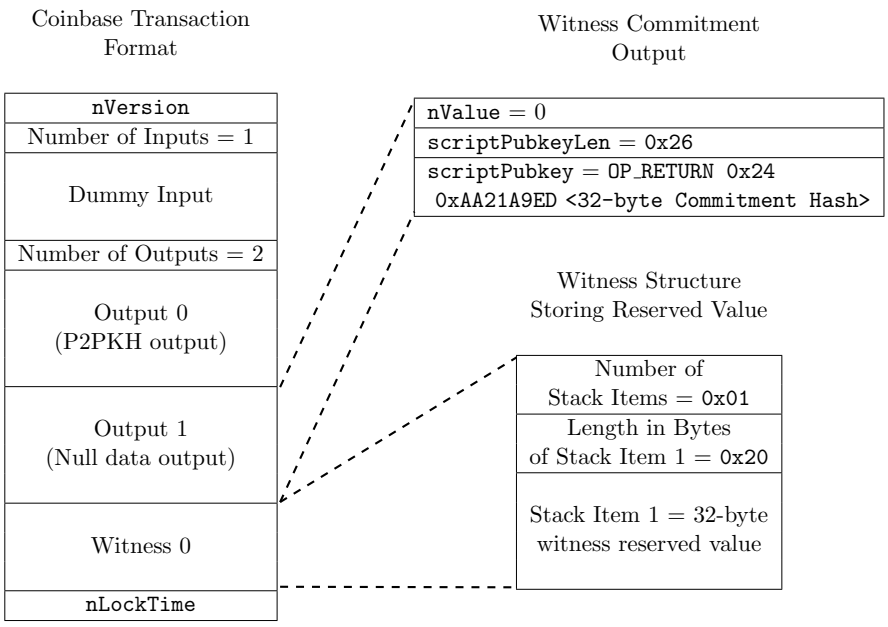


Figure 5.27: Example of a coinbase transaction for a block with SegWit transactions

As the coinbase transaction outputs are included in its TXID calculation, any tampering with the signature data in the witness structures will modify the TXID of the coinbase transaction. This in turn will change the `hashMerkleRoot` in the block header. Thus modifying the witness structures in a valid block will invalidate it. This indirect method for ensuring the integrity of the witness structures is necessary due to the need to maintain compatibility with pre-SegWit clients. Integrity of the witness structures could have been ensured by including a new field in the block header containing the witness commitment hash. But this is a hard fork which would require all the nodes in the network to upgrade to SegWit-capable client software.

Pre-SegWit clients do not receive the witness structures associated with SegWit inputs. When these clients receive a block whose coinbase transaction has an output with a witness commitment, they will interpret the output as a null data output and ignore it. On the other hand, SegWit-capable clients will check that the witness commitment hash is correct using the WTXID Merkle root hash and the witness reserved value.

5.10 SegWit Signature Generation

The messages which are hashed to generate pre-SegWit signatures for each input typically have several fields in common. For example, consider the messages shown in Figure 5.16 corresponding to the two inputs of the transaction

shown in Figure 5.15. The fields corresponding to the two transaction outputs appear in both the messages. The fields containing the TXID and index of the previous outputs (`hash0`, `n0`, `hash1`, `n1`) being unlocked by the inputs appear in both the messages. The sequence number fields of the inputs (`nSequence0`, `nSequence1`) appear in both the messages. The consequence of such repetitions is that the amount of data to be hashed by the double SHA-256 hash function increases quadratically, i.e. $O(N^2)$, with the number of inputs N. This is undesirable as this quadratic complexity is due to an oversight in the original design. The amount of data hashed can be made to increase only linearly with the number of inputs if the fields which are repeatedly hashed are hashed only once and the hash values reused. Such a modification of the message digest calculation will require a hard fork change in the Bitcoin protocol which entails upgrading all the nodes in the network. For this reason, the message digest calculation was not changed. But when SegWit was proposed as a soft fork solution to transaction malleability, the Bitcoin developers saw an opportunity to introduce a more efficient message digest calculation algorithm for the signatures in SegWit inputs. As pre-SegWit clients see SegWit outputs as anyone-can-spend outputs, they do not need to calculate the signatures required to validate the SegWit inputs which unlock these outputs. SegWit-capable clients use the new message digest calculation algorithm for validating SegWit inputs and the old message digest calculation algorithm for validating non-SegWit inputs.

Figure 5.28 shows the messages used to generate the signatures in each of the two inputs of a regular transaction. The message always contains the `nVersion` field, the `nLockTime` field, and the `nHashType` field. Three new fields `hashPrevouts`, `hashSequence`, and `hashOutputs` not present in pre-SegWit signature messages are also always included in the message. The signature message corresponding to a particular input includes six fields related to that input. They are as follows:

1. The TXID (`hash0` or `hash1`) of the transaction containing the output being unlocked.

2. The index (`n0` or `n1`) of the output being unlocked in the transaction containing it.

3. The length of the challenge script (`prevScriptPubkeyLen0` or `prevScriptPubkeyLen1`) from the output being unlocked.

4. The challenge script itself (`prevScriptPubkey0` or `prevScriptPubkey1`).[17]

5. The `nValue` field from the output being unlocked (`prevNValue0` or `prevNValue1`) which contains the amount of bitcoins associated with

[17] To keep the exposition simple, we once again assume that the challenge script does not contain the `OP_CODESEPARATOR` operator.

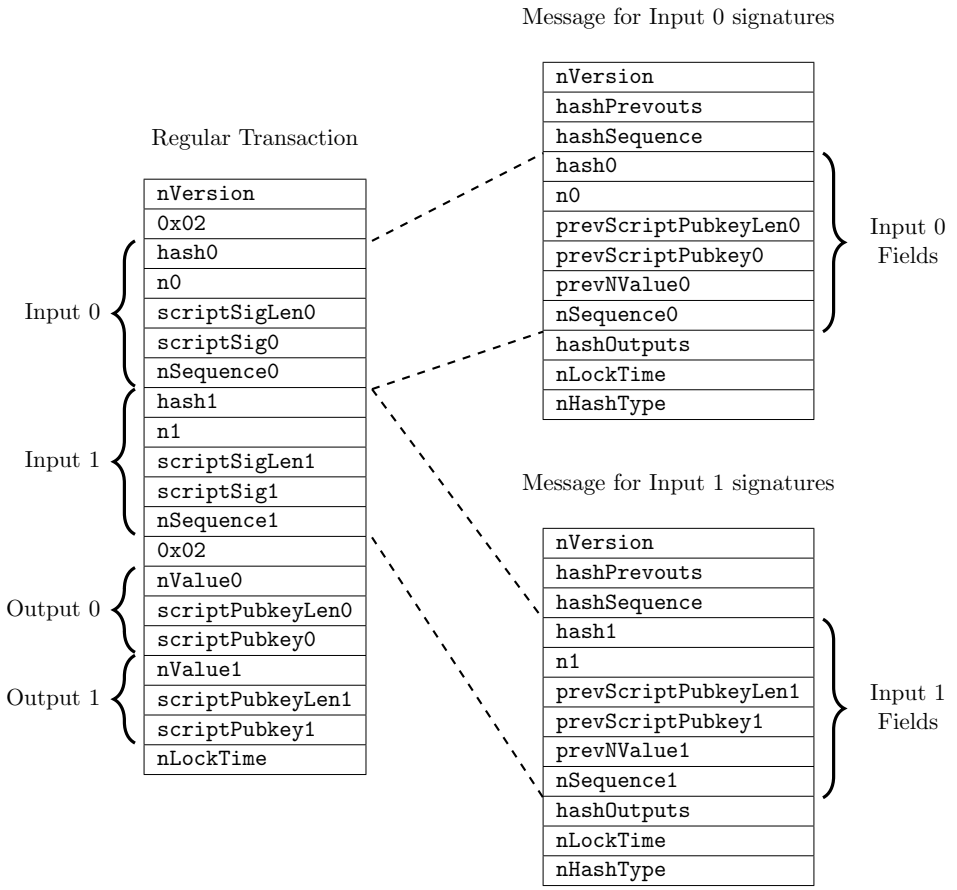


Figure 5.28: Messages used to generate SegWit signatures for the both inputs of a transaction with two inputs

the output. This field was not present in any of the messages used in the pre-SegWit signature generation. The motivation for including this field was to make the offline signing of transactions by devices which do not have access to the whole blockchain safer. When an untrusted entity requests an offline device to sign a transaction, the presence of the amount in the message ensures that the signature becomes invalid if the amount was misrepresented.

6. The sequence number field of the input (`nSequence0` or `nSequence1`) is always included.

The `hashPrevouts`, `hashSequence`, and `hashOutputs` fields are defined according to the signature hash type as shown in Table 5.5. In the table and description below, we have omitted the `SIGHASH_` prefix of the hash types and

Signature Hash Type	**Field Definitions**
`ALL`	`hashPrevouts` = SHA256d(`hash0` ‖ `n0` ‖ `hash1` ‖ `n1`) `hashSequence` = SHA256d(`nSequence0` ‖ `nSequence1`) `hashOutputs` = SHA256d(`nValue0` ‖ `scriptPubkeyLen0` ‖ `scriptPubkey0` ‖ `nValue1` ‖ `scriptPubkeyLen1` ‖ `scriptPubkey1`)

`NONE`	`hashPrevouts` = SHA256d(`hash0` ‖ `n0` ‖ `hash1` ‖ `n1`)
	`hashSequence` = `0x0000...0000`
	`hashOutputs` = `0x0000...0000`

`SINGLE`	`hashPrevouts` = SHA256d(`hash0` ‖ `n0` ‖ `hash1` ‖ `n1`) `hashSequence` = `0x0000...0000` `hashOutputs` = SHA256d(`nValue0` ‖ `scriptPubkeyLen0` ‖ `scriptPubkey0`) or SHA256d(`nValue1` ‖ `scriptPubkeyLen1` ‖ `scriptPubkey1`)

`ALL|`
`ANYONECANPAY`
	`hashPrevouts` = `0x0000...0000`
	`hashSequence` = `0x0000...0000`
	`hashOutputs` =
	SHA256d(`nValue0` ‖ `scriptPubkeyLen0` ‖ `scriptPubkey0` ‖ `nValue1` ‖ `scriptPubkeyLen1` ‖ `scriptPubkey1`)

`NONE|`
`ANYONECANPAY`
	`hashPrevouts` = `0x0000...0000`
	`hashSequence` = `0x0000...0000`
	`hashOutputs` = `0x0000...0000`

`SINGLE|`
`ANYONECANPAY`
	`hashPrevouts` = `0x0000...0000`
	`hashSequence` = `0x0000...0000`
	`hashOutputs` =
	SHA256d(`nValue0` ‖ `scriptPubkeyLen0` ‖ `scriptPubkey0`)
	or
	SHA256d(`nValue1` ‖ `scriptPubkeyLen1` ‖ `scriptPubkey1`)

Table 5.5: Definitions of the `hashPrevouts`, `hashSequence`, and `hashOutputs` fields used to generate the SegWit signature messages for the regular transaction in Figure 5.28.

use SHA256d(·) to denote the SHA256d hash function. The following rules are used to define the fields:

1. If the `ANYONECANPAY` hash type is used in conjunction with any of the other three base hash types, the `hashPrevouts` field is set to the 256-bit all zeros bitstring. Otherwise, `hashPrevouts` field is set to the SHA256d hash of the concatenation of the TXIDs and indices of the outputs being unlocked by the transaction. The rationale is that if the `ANYONECANPAY` hash type is specified then all the outputs being unlocked may not be known at the time of signature generation for a particular input.

2. If the hash type is `ALL`, the `hashSequence` field is to the SHA256d hash of

all the sequence number fields from the inputs. Otherwise, it is set to the 256-bit all zeros bitstring. This definition of `hashSequence` follows from the fact that all the sequence numbers are included in the pre-SegWit messages only when the hash type is `ALL`.

3. If the hash type is `ALL` or `ALL|ANYONECANPAY`, the `hashOutputs` field is set to the SHA256d hash of the concatenation of all the output fields. If the hash type is `SINGLE` or `SINGLE|ANYONECANPAY`, the `hashOutputs` field is set to the SHA256d hash of the output which has the same index as the input being signed. For example, messages for Input 0 with `SINGLE` hash type will have a `hashOutputs` field equal to

   SHA256d(`nValue0` ‖ `scriptPubkeyLen0` ‖ `scriptPubkey0`).

   If the hash type is `NONE` or `NONE|ANYONECANPAY`, the `hashOutputs` field is set to the 256-bit all zeros bitstring. So `hashOutputs` either contains a hash of all the outputs, a single output, or none of the outputs.

These three fields help reduce the amount of data which is hashed if the hash types of the inputs result in identical field values. For example, in Figure 5.28 suppose both Input 0 and Input 1 have signatures of hash type `ALL`. The the values of the `hashPrevouts`, `hashSequence`, and `hashOutputs` fields are the same for both the inputs. These fields will calculated once for Input 0 and reused for Input 1.

5.11 Block Size and Sigop Limits

Prior to SegWit activation, the size of single block could be at most 1 MB (10^6 bytes). Additionally, the number of *signature operations (sigops)* in a block could be at most 20,000. The operators which validate ECDSA signatures, i.e. `OP_CHECKSIG`, `OP_CHECKSIGVERIFY`, `OP_CHECKMULTISIG`, `OP_CHECKMULTISIGVERIFY`, contribute to the sigop count. The `OP_CHECKSIG` and `OP_CHECKMULTISIG` operators were discussed in Section 5.5 in the context of P2PK, P2PKH, and multisig challenge scripts. The `OP_CHECKSIGVERIFY` operator does the same operation as the `OP_CHECKSIG` operator and then checks that the top stack element evaluates to `True`. The `OP_CHECKMULTISIGVERIFY` is a similar extension of the `OP_CHECKMULTISIG` operator. The limit on sigops is to prevent malicious nodes from causing CPU exhaustion attacks on the network by broadcasting blocks with a large number of signature validation operations. The number of sigops in a block is calculated as follows:

1. Each `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` operator in the `scriptPubkey` and `scriptSig` fields in the transactions contained in the block is counted as one sigop. For example, the below P2PKH challenge script contributes only one sigop corresponding to the `OP_CHECKSIG` operator.

   `OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`.

2. Each `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operator in the `scriptPubkey` and `scriptSig` fields in the transactions contained in the block is counted as 20 sigops. The number 20 is the maximum number for public keys (the number n in a m-of-n multisig script) which these operators accept. Taking this maximum number instead of the actual number of public keys was probably done to keep the implementation simple. It gives a pessimistic estimate of the number of sigops in a block. For example, the below 1-of-2 multisig challenge script contributes 20 sigops.

   `OP_1 <PubKey1> <PubKey2> OP_2 OP_CHECKMULTISIG`.

3. If the `scriptSig` field of a transaction contains a response to a P2SH challenge script, then the redeem script contributes sigops according to the following rules:

   (i) Each `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` operator in the redeem script is counted as one sigop.

   (ii) Each `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operator in the redeem script preceeded by an `OP_n` operator with $n \in \{1, 2, \ldots, 16\}$ is counted as n sigops.

   (iii) `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operators not preceeded by an `OP_n` operator are counted as 20 sigops each.

   For example, the below P2SH response script with a redeem script consisting of a 1-of-2 multisig script contributes 2 sigops. We have explicitly indicated the push of the N-byte redeem script by the `PushN` operator.

   `OP_0 <Signature 1> <PushN>` $\underbrace{\texttt{< OP\_1 <PubKey1> <PubKey2> OP\_2 OP\_CHECKMULTISIG >}}_{\text{Redeem Script of Length } N \text{ bytes}}$.

   Note that the `OP_CHECKMULTISIG` in the redeem script does not contribute 20 sigops (according to rule 2) inspite of being in a `scriptSig` field. This is because when the operators in the P2SH response script are parsed the `OP_CHECKMULTISIG` in the redeem script is considered part of the N-byte data which is pushed onto the stack. It is not interpreted as an operator in the `scriptSig` field.

SegWit increased both the block size and sigop limits for blocks containing SegWit transactions while keeping these limits the same for blocks without SegWit transactions. We discuss the block size increase first followed by the sigop limit increase.

Block Weight Limit

Recall that there are two serializations of a transaction as shown in Figure 5.24. The first serialization does not include the witness structures while the second serialization does not include them. Both these serializations are identical for non-SegWit transactions which do not contain any SegWit inputs. Let the *base size* of a block be the size of the block when all its transactions (SegWit or not) are serialized according to the first serialization. Let the *total size* of a block be the size of the block when all its transactions are serialized according to the second serialization. Define the *block weight* as follows:

$$\text{Block Weight} = 3 \times \text{Base Size} + \text{Total Size}.$$

SegWit imposes the restriction

$$\text{Block Weight} \leq 4 \text{ MB} = 4 \times 10^6 \text{ bytes}$$

Note that the total size of a block is equal to the base size plus the size of the witness structures in it (let us call the latter *witness size*). So the block weight restriction can be rewritten as

$$4 \times \text{Base Size} + \text{Witness Size} \leq 4 \text{ MB} \implies \text{Base Size} + \frac{\text{Witness Size}}{4} \leq 1 \text{ MB}.$$

The above inequality implies that the base size cannot exceed 1 MB irrespective of the witness size. This design was intentional as SegWit is a soft fork change to the Bitcoin protocol. If a block with SegWit transactions satisfies the block weight restriction, it will also satisfy the pre-SegWit block size restriction. So the block will be considered valid by nodes running pre-SegWit clients. By scaling down the witness size by a factor of four, SegWit allows for an increase in the number of transactions which can be included in a block if they follow the SegWit script templates. As the actual size of the block which is added to the blockchain is sum of its base size and witness size, the block size limit exceeds the previous limit of 1 MB. The factor four in the block weight calculation was chosen to constrain the increase to a moderate amount.

After SegWit activation, the *virtual size* of a transaction is used to calculate the fee rate. Let the *base transaction size* and the *total transaction size* be length of a transaction in bytes given by the first and second serializations in Figure 5.24 respectively. The virtual size of a transaction is given by

$$\frac{3 \times \text{Base Tx Size} + \text{Total Tx Size}}{4} = \text{Base Tx Size} + \frac{\text{Tx Witness Size}}{4}$$

where fractional values are rounded up to the next integer. The fee rate of the transaction is then given by the fees divided by its virtual size. For non-SegWit transactions, the virtual size is equal to the base transaction size since there are no witness structures in them. So the fee rate calculation remains the same for these transactions.

Sigop Limit

SegWit increases the limit on the number of sigops in a block to 80,000. The number of sigops in a block is calculated as follows where the first three rules are the same as the pre-SegWit sigop rules with a scaling factor of four.

1. Each `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` operator in the `scriptPubkey` and `scriptSig` fields in the transactions contained in the block is counted as four sigops.

2. Each `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operator in the `scriptPubkey` and `scriptSig` fields in the transactions contained in the block is counted as 80 sigops.

3. If the `scriptSig` field of a transaction contains a response to a P2SH challenge script, then the redeem script contributes sigops according to the following rules:

   (i) Each `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` operator in the redeem script is counted as four sigops.

   (ii) Each `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operator in the redeem script preceeded by an `OP_n` operator with $n \in \{1, 2, \ldots, 16\}$ is counted as $4n$ sigops.

   (iii) `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operators not preceeded by an `OP_n` operator are counted as 80 sigops each.

4. Each P2WPKH input in a transaction contributes one sigop. Recall that a P2WPKH script template has challenge and response scripts given by

   ```
   scriptPubkey:  OP_0 0x14 <PubKeyHash>,
      scriptSig:  (empty),
   scriptWitness: <Signature> <Public Key>.
   ```

   A transaction with a P2WPKH input has the `scriptSig` and `scriptWitness` fields in its serialization. Even there are no signature validation operators explicitly appearing in these fields, the validation of this input involves a `OP_CHECKSIG` operation (see Section 5.8).

5. P2SH-P2WPKH inputs also contribute one sigop each.

6. If an input is a P2WSH input, then operators in the P2WSH redeem script contribute sigops according to the following rules:

   (i) Each `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` operator is counted as one sigop.

(ii) Each `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operator preceeded by an `OP_n` operator with $n \in \{1, 2, \ldots, 16\}$ is counted as n sigops.

(iii) `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` operators not preceeded by an `OP_n` operator are counted as 20 sigops each.

7. P2SH-P2WSH inputs contribute the same sigops as a P2WSH input with the `scriptWitness` field.

For a non-SegWit input, rules 4 through 7 do not apply. The sigop count is exactly four times the pre-SegWit sigop count. Since the sigop limit is also four times the previous limit, a non-SegWit block which has at most 20,000 sigops will continue to be valid under the SegWit sigop calculation rules.

Chapter 6

Contracts

In spite of its limited functionality, the Bitcoin scripting language can be used to create contracts which encode conditional exchange of bitcoins among entities. The entities participating in a contract typically signal their intent by signing a transaction or by providing a valid response script to unlock a UTXO. Some contracts have safeguards protecting honest entities against malicious or uncooperative behaviour other participants. These safeguards either enable complete recovery of the bitcoins invested by the honest entities while entering the contract. While there are several contracts possible, we discuss three examples: *escrow*, *micropayments*, and *decentralized lotteries*.

6.1 Escrow

Consider the scenario where Alice (the buyer) wants to purchase a used book from Bob (the seller) using bitcoins. Alice and Bob live in different cities making it infeasible for them to meet and perform the transaction. Bob promises to ship the book to Alice once he receives the bitcoin payment. But Alice does not trust Bob and fears that he may not send her the book after receiving the payment. To reduce her risk, Alice proposes to use an escrow contract to pay Bob. The contract needs a third party Carol (the escrow) who both Alice and Bob trust. The contract proceeds as follows:

1. Alice requests public keys from Bob and Carol. Let these keys be `PubKeyB` and `PubKeyC` respectively.
2. Alice transfers x bitcoins to a 2-of-3 multisig output which has the challenge script

   ```
   OP_2 <PubKeyA> <PubKeyB> <PubKeyC> OP_3 OP_CHECKMULTISIG
   ```

   where `PubKeyA` is Alice's public key.
3. Once Bob sees that Alice's transaction has appeared on the blockchain, he ships the book to Alice.

4. The funds locked in the multisig output can be spent if any two of Alice, Bob, and Carol provide signatures created by their respective private keys. Any of the three following scenarios can happen.

   (i) Alice is happy with the book she has received. She signs a transaction which unlocks the 2-of-3 multisig output and transfers the x bitcoins (minus the transaction fees) to the P2PK address containing Bob's public key. She sends this transaction to Bob who adds his own signature and broadcasts it on the network for inclusion on the blockchain.

   (ii) Alice receives the book but refuses to sign the transaction paying Bob. Bob provides proof of shipment to the escrow Carol and requests her to sign a transaction paying him. If Carol is convinced that Bob actually shipped the book to Alice, she will send the signed transaction to Bob, who will add his own signature to the transaction and broadcast it on the network.

   (iii) Bob does not ship the book to Alice. Furthermore, he refuses to sign the transaction refunding the bitcoins to Alice. In this case, Alice requests Carol to sign a transaction refunding the bitcoins. If Carol complies, Alice adds her own signature to the refund transaction and broadcasts it on the network.

The above escrow contract fails if the escrow Carol colludes with Alice or Bob. If Alice and Carol collude, then they can refuse to pay Bob even if he sent the book to Alice. If Bob and Carol collude, then they can transfer the bitcoins to any address without sending the book to Alice. Another weakness of the contract is that it is difficult for Bob to give proof of shipment. He can send the tracking information of the package to Carol but the package itself may be empty. A solution to the empty package problem is to choose an escrow Carol who lives in the same city as Alice, ask Bob to ship the book to Carol, and have Alice collect it from her. Alice can open the package in Carol's presence and Carol can verify that it is not empty.

6.2 Micropayments

Even Bitcoin transaction involves paying transaction fees which makes using Bitcoin to make small payments expensive (the transaction fees may exceed the payment amount). But if a sequence of small payments are to be made to the same entity, the micropayment contract can be used which aggregates the small payments and requires that transaction fees be paid for only one transaction.

Consider the scenario where Alice offers proofreading and editing services online in return for bitcoins. Clients can email Alice their documents and

Alice will reply with typos and grammatical errors she has found in the documents. Alice charges her clients a fixed amount of bitcoins per edited page. To avoid the situation where a client refuses payment after receiving the edited document, Alice uses the micropayment contract. This contract enables her to get payment incrementally for each page she edits. Let Bob be a client who wishes a to have a 100 page document proofread by Alice. Let us assume that Alice charges 0.0001 bitcoins per page. So Bob expects to pay a maximum of 0.001 bitcoins to Alice. The protocol proceeds as follows:

1. Bob requests a public key from Alice. He also generates a public key for himself. Let `PubKeyA` and `PubKeyB` be Alice's and Bob's public keys respectively.

2. Bob creates a transaction t_1 which transfers 0.01 bitcoins to a 2-of-2 multisig output which has the challenge script

   `OP_2 <PubKeyA> <PubKeyB> OP_2 OP_CHECKMULTISIG`.

   Bob does not broadcast t_1 on the network at this point. If he does, then he is liable to have his funds locked in the multisig output forever in the event that Alice refuses to sign any transaction which spends this output.

3. Bob creates a refund transaction t_2 which using the TXID of t_1 which unlocks the multisig output in t_1 and transfers the 0.01 bitcoins (minus transaction fees) to an address owned by him. A relative lock time of n days is set on t_2. This prevents t_2 from being broadcast in the network until n days have passed after t_1 was included on the blockchain. At this point, the response script in t_2's input has no signatures.

4. Bob includes his signature in the response script in t_2 and sends it to Alice. He asks her to add her signature in the response script and return t_2 to him. If Alice refuses to do this, Bob can terminate the contract. He has not lost any funds as t_1 has not been recorded on the blockchain.

5. If Alice includes her signature in t_2 and returns it to Bob, both Alice and Bob have fully signed the refund transaction which spends the 2-of-2 multisig output in t_1 to refund Bob.

6. Bob now broadcasts t_1 on the network. Once it is included on the blockchain, he sends Alice his document.

7. Alice edits only the first page of the document. She creates a transaction e_1 which unlocks the 2-of-2 multisig output in t_1 and pays her 0.0001 bitcoins and the remaining 0.0099 bitcoins (minus transaction fees) to Bob.

8. Alice includes her signature in e_1 and sends it to Bob along with the first page edits.

   (i) If Bob refuses to sign e_1, then Alice is unpaid only for the effort spent in editing one page. She terminates the contract. Bob broadcasts the refund transaction t_2 after the relative lock time expires and receives the 0.01 bitcoins (minus transaction fees).

   (ii) If Bob signs e_1 and returns it to Alice, then Alice is guaranteed at least 0.0001 bitcoins if she broadcasts e_1 before the relative lock time on t_2 expires. But Alice does not broadcast e_1 at this point.

9. Alice edits the second page of the document. She creates a transaction e_2 which unlocks the 2-of-2 multisig output in t_1 and pays her 0.0002 bitcoins and the remaining 0.0098 bitcoins (minus transaction fees) to Bob.

10. Alice includes her signature in e_2 and sends it to Bob along with the second page edits.

    (i) If Bob refuses to sign e_2, then Alice can broadcast e_1 and get paid for the edits in the first page. She is unpaid only for the effort spent in editing the second page. She terminates the contract. When Alice broadcasts e_1, Bob receives 0.0099 bitcoins (minus transaction fees).

    (ii) If Bob signs e_2 and returns it to Alice, then Alice is guaranteed at least 0.0002 bitcoins if she broadcasts e_2 before the relative lock time on t_2 expires. But Alice does not broadcast e_2 at this point.

11. Alice continues this process of sending edits for the next page along with a transaction requesting cumulative payment for all pages edited so far. Once all the pages have been edited, the contract terminates. Figure 6.1 illustrates the steps in the protocol when neither Alice nor Bob cheats. Alice has to take care to finish editing before the relative lock time on t_2 expires. So she has n days after t_1 is confirmed to finish the edits.

If Bob refuses to sign any of the e_i transactions, Alice will not edit the subsequent pages. But Bob can always cheat Alice out of the payment for the last page (page 100) as he receives the edits for the last page along with a request to sign e_{100}. This risk should be acceptable to Alice as she anyway receives payment for the first 99 pages. If Alice wants to avoid not getting paid for the last page, she can distribute the cost of editing the last page across the cost of editing the first 99 pages and offer the last page edits for free.

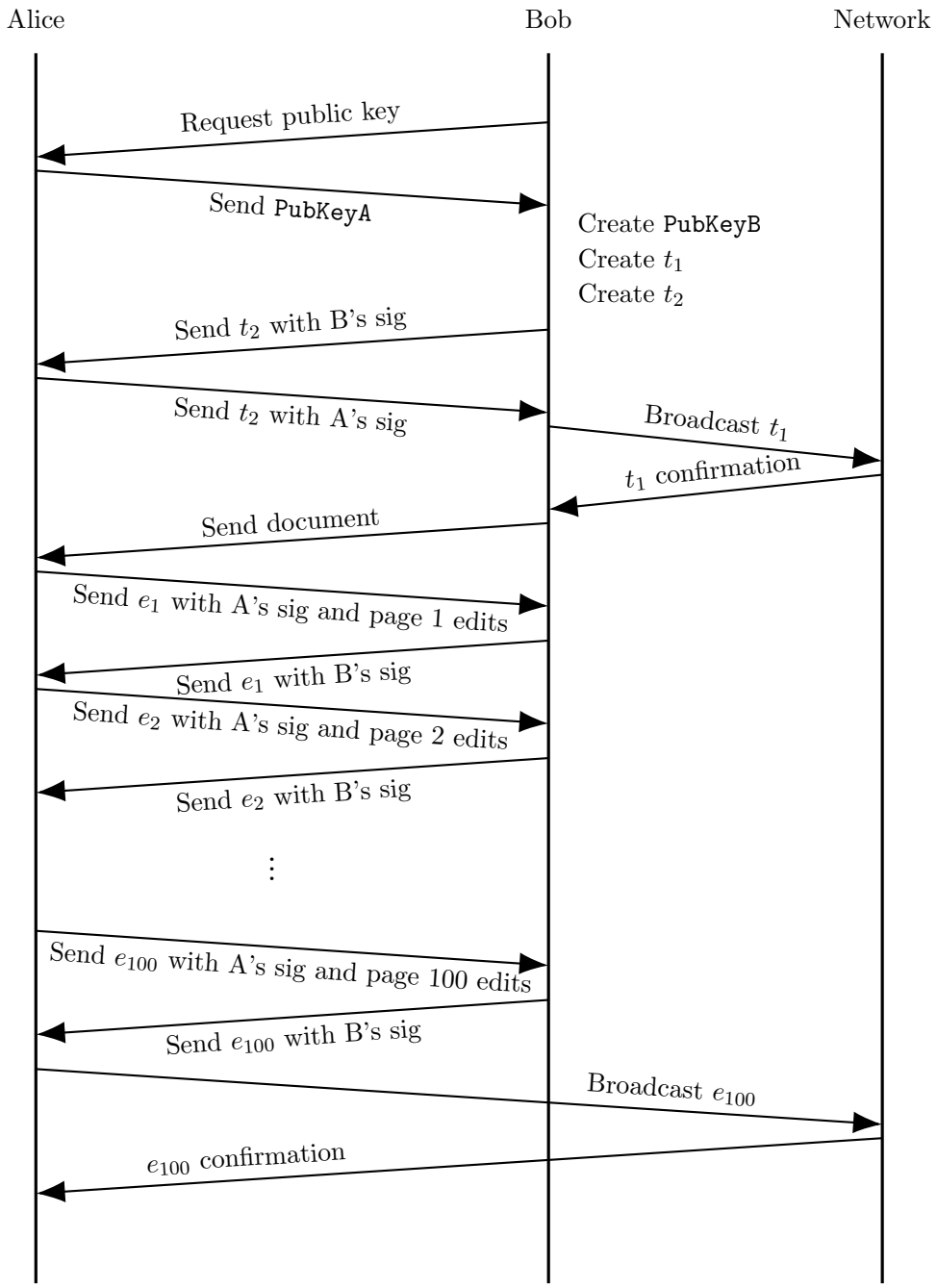


Figure 6.1: Illustration of the steps in the micropayments protocol when neither Alice nor Bob cheats

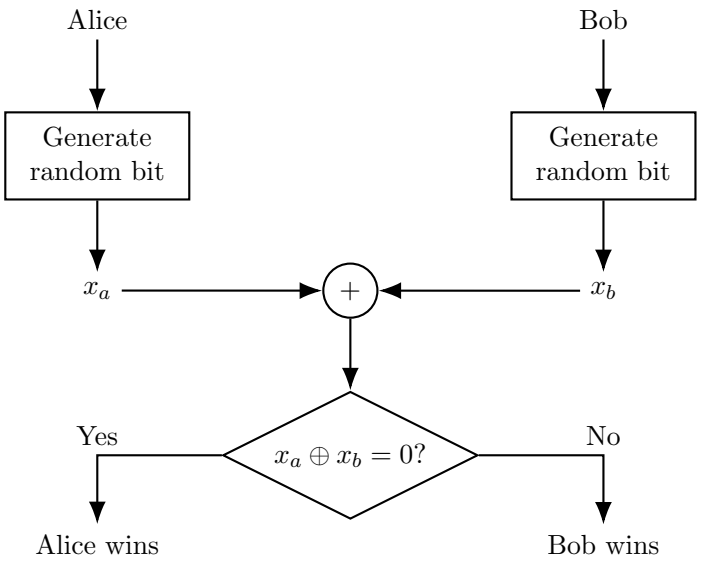


Figure 6.2: High-level description of the two-person lottery protocol

6.3 Decentralized Lotteries

In a traditional lottery, participants purchase lottery tickets from the organizer of the lottery. Some part of the money collected from the ticket sale will go to the organizer for organizing the lottery and the rest will go to the jackpot. The organizer picks a winner at random and awards him/her the jackpot. The organizer can cheat in two ways. Firstly, he can charge an unreasonable fee for organizing the lottery. Secondly, he can manipulate the procedure used to pick the lottery winner and award the jackpot to a ticket holder of his choice. These problems are eliminated in a Bitcoin-based decentralized lottery, i.e. a lottery without an organizer.

For ease of exposition, we will first consider the case of a lottery with only two participants, Alice and Bob. Let us assume that Alice and Bob each put in one bitcoin. One of them needs to be randomly chosen to be the winner who receives two bitcoins. At a high level, the lottery protocol involves both Alice and Bob generating a single random bit each. Let these bits be x_a and x_b. If the XOR of the bits is zero, Alice is the winner. Otherwise, Bob is the winner. This protocol is illustrated in Figure 6.2. If a trusted third party were available, Alice and Bob could send their bits and bitcoins to the third party who would calculate the XOR of the bits and give the winner the two bitcoins. In a decentralized setup, there is no trusted third party and messages need to be exchanged between Alice and Bob. If Alice sends her bit x_a to Bob first, then Bob can claim that his bit x_b is the complement of x_a resulting in $x_a \oplus x_b = 1$. If Bob sends his bit to Alice first, Alice can cheat in a similar manner. The solution is to use the blockchain to compel Alice and Bob to

commit to their bits at the beginning of the lottery. The protocol proceeds as follows:

1. Alice chooses a random bytestring `SecretA` of length either 16 or 17 bytes which she keeps a secret. She calculates the SHA-256 hash `HashA` of `SecretA` and sends it to Bob. By revealing only the hash `HashA` to Bob, Alice commits to the secret bytestring `SecretA`. The length of `SecretA` represents the random bit chosen by Alice. The lengths 16 bytes (128 bits) and 17 bytes (136 bits) were chosen to make it infeasible for Bob to recover `SecretA` from `HashA` by brute force search. Any two distinct lengths which are long enough to prevent brute force attacks can be used.

2. Bob also chooses a random bytestring `SecretB` of length either 16 or 17 bytes. He calculates its SHA-256 hash `HashB` and sends it to Alice.

3. The successful execution of the protocol requires Alice and Bob to reveal their secret bytestrings. To ensure this, Alice broadcasts a *deposit transaction* on the network which unlocks a UTXO she owns and pays two bitcoins to an output locked by a P2SH challenge script with redeem script given below (indented for readability).

```
OP_IF
     OP_SHA256 <HashA> OP_EQUALVERIFY <PubKeyA>
OP_ELSE
     <Timeout> OP_CHECKLOCKTIMEVERIFY OP_DROP <PubKeyB>
OP_ENDIF
OP_CHECKSIG
```

   The `OP_IF` operator pops the the top stack element and checks if it evaluates to `True`. If yes, the script between the `OP_IF` operator and the `OP_ELSE` operator is executed. Otherwise, the script between the `OP_ELSE` operator and the `OP_ENDIF` operator is executed. Note that the `OP_CHECKSIG` operator is executed at the very end in both cases. The `OP_CHECKLOCKTIMEVERIFY` operator checks that the top stack element is less than the `nLockTime` field of the transaction which unlocks the output of the deposit transaction. If yes, the script execution continues. Otherwise, it terminates. As transactions with a lock time enabled cannot be included in a block until the lock time expires, the `OP_CHECKLOCKTIMEVERIFY` operator ensures that the deposit transaction output is not spent until the after the block height or Unix time specified in the `Timeout` field has been reached.

   There are two possible response scripts to the above redeem script. Alice can spend the output in the deposit transaction at any time by providing

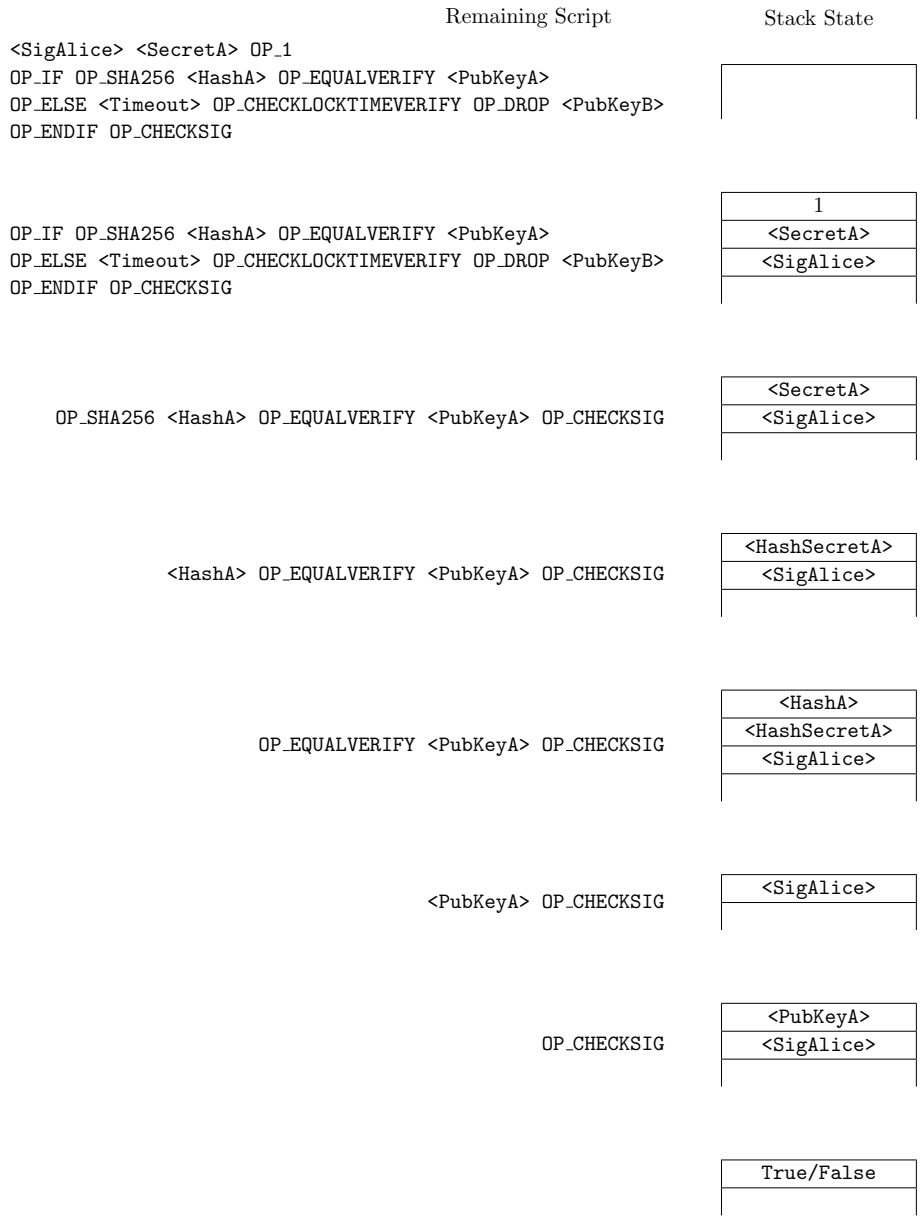


Figure 6.3: Stack state during the execution of the deposit transaction redeem script given Alice's response

a response of the form

```
<SigAlice> <SecretA> OP_1
```

where `SigAlice` is Alice's signature. Figure 6.3 shows the state of the stack during the execution of the redeem script given Alice's response

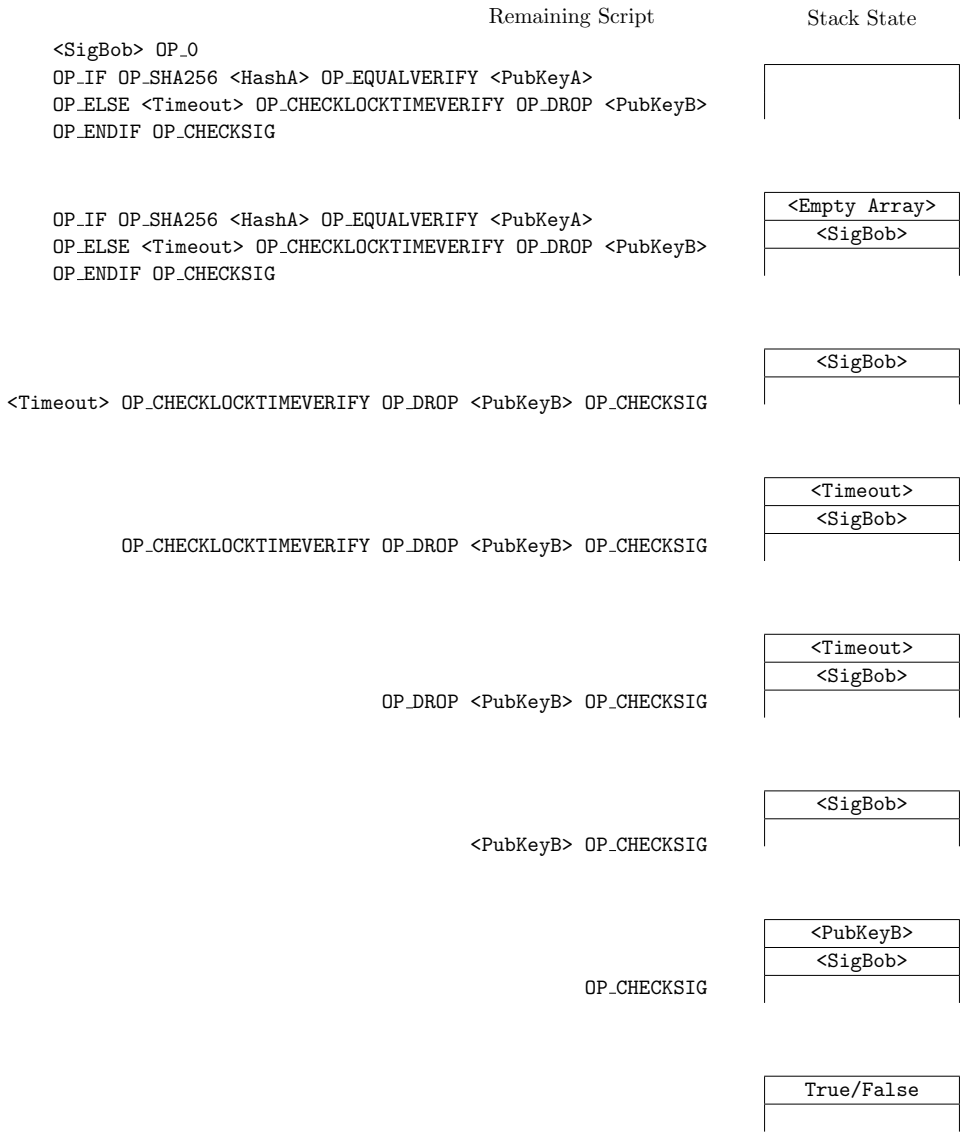


Figure 6.4: Stack state during the execution of the deposit transaction redeem script given Bob's response

to it. Alternatively, if the current block height or Unix time exceeds the timeout encoded in the `Timeout` field then Bob can spend the output in the deposit transaction by providing a response of the form

`<SigBob> OP_0`.

Figure 6.4 shows the state of the stack during the execution of the redeem script given Bob's response to it. Recall that the `OP_0` operator pushes an empty byte array onto the stack which evaluates to `False`.

To summarize, Alice can unlock the deposit transaction output at any time by providing the secret bytestring `SecretA` which hashes to `HashA`. When the transaction which spends the output appears on the blockchain, `SecretA` is revealed to Bob. If Alice does not spend the deposit transaction output before `Timeout` is exceeded, then Bob can spend the output by providing a signature. Since the deposit output contains two bitcoins which originally belonged to Alice, she will incur a loss of two bitcoins if she does not reveal her secret before the timeout.

4. Bob also broadcasts a deposit transaction on the network which unlocks a UTXO he owns and pays two bitcoins to an output locked by a P2SH challenge script with redeem script given below.

```
OP_IF
     OP_SHA256 <HashB> OP_EQUALVERIFY <PubKeyB>
OP_ELSE
     <Timeout> OP_CHECKLOCKTIMEVERIFY OP_DROP <PubKeyA>
OP_ENDIF
OP_CHECKSIG
```

The above script is complementary to the one used by Alice in her deposit transaction. Bob can unlock his deposit transaction output at any time by providing the secret bytestring `SecretB` which hashes to `HashB`. When the transaction which spends the output in this way appears on the blockchain, `SecretB` is revealed to Alice. If Bob does not spend the deposit transaction output before `Timeout` is exceeded, then Alice can spend the output by providing a signature.

5. Alice and Bob wait for both the deposit transactions to be confirmed on the blockchain.

6. Let us assume that Alice and Bob can both unlock UTXOs each containing at least one bitcoin. Alice creates a transaction which unlocks these UTXOs and pays two bitcoins to an output which has a P2SH challenge script that can be unlocked by a response script[1] having *any one* of the following two forms:

   (i) If the lengths of `SecretA` and `SecretB` are equal, the response to the redeem script contains Alice's signature followed by the bytestrings `SecretA` and `SecretB`. The `scriptSig` field is given by

   $$\underbrace{\texttt{<SigAlice> <SecretA> <SecretB>}}_{\text{Response to redeem script}}\ \texttt{<Redeem Script>}.$$

[1] Recall that a valid response script corresponding to a P2SH challenge script consists of the response to a redeem script followed by the redeem script itself.

(ii) If the lengths of `SecretA` and `SecretB` are not equal, the response to the redeem script contains Bob's signature followed by the bytestrings `SecretA` and `SecretB`. The `scriptSig` field is given by

$$\underbrace{\texttt{<SigBob> <SecretA> <SecretB>}}_{\text{Response to redeem script}}\ \texttt{<Redeem Script>}.$$

While Alice and Bob know their own secret bytestrings, they need the other secret bytestring to construct a valid response script. We will discuss how they acquire the other bytestring below.

The redeem script consists of three functional parts shown below.

$$\texttt{<Redeem Script>} = \texttt{<Check Hashes> <Compute Winner> <Check Sig>}.$$

The `<Check Hashes>` portion of the redeem script checks that the bytestrings given in the response script hash to `HashA` and `HashB`. In Script notation, it is given by

```
OP_2DUP OP_SHA256 <HashB> OP_EQUALVERIFY OP_SHA256 <HashA> OP_EQUALVERIFY
```

where the `OP_2DUP` operator duplicates the top two stack elements, the `OP_SHA256` operator replaces the top stack element with its SHA-256 hash, and the `OP_EQUALVERIFY` operator pops the top two stack elements and checks them for equality (if they are equal script execution proceeds, otherwise it terminates). Figure 6.5 shows the state of the stack during the execution of the `<Check Hashes>` portion of the redeem script. We assume that the response to the redeem script (with Alice's signature) has already been pushed onto the stack. The stack items `HashSecretA` and `HashSecretB` represent the SHA-256 hashes of the bytestrings `SecretA` and `SecretB` respectively. The two `OP_EQUALVERIFY` operators check that these items are equal to the `HashA` and `HashB` bytestrings provided in the redeem script. If either of the secret bytestrings provided in the redeem script do not have the required hashes, the script execution terminates and the remaining portion of the redeem script is not executed. As the stack can only store byte arrays, the secrets `SecretA` and `SecretB` were chosen to be bytestrings of length either 16 or 17 bytes. If the stack had allowed storage of arbitrary bitstrings, we could have chosen the secrets to be bitstrings of length 128 or 129 bits.

If the `<Check Hashes>` portion of the redeem script succeeds, the script execution proceeds with the `<Compute Winner>` portion which compares the lengths of `SecretA` and `SecretB`. If the lengths are equal, then Alice is the winner. Otherwise, Bob is the winner. This is akin to the bitwise XOR to decide the winner where Alice won if the bits were equal and

Bob won otherwise. After the execution of `<Check Hashes>`, the top stack element is set to 0 to indicate that Alice is the winner and set to 1 to indicate that Bob is the winner. In Script notation, `<Compute Winner>` is given by

```
OP_SIZE OP_ROT OP_SIZE OP_NIP OP_EQUAL
```

where the `OP_SIZE` operator pushes the length of the top stack element in bytes onto the stack, the `OP_ROT` operator cyclically rotates the top three stack elements once, and the `OP_NIP` operator deleletes the stack item below the top stack element.[2] Figure 6.6 shows the state of the stack during the execution of the `<Compute Winner>` portion of the redeem script. The `<Check Sig>` portion of the redeem script checks the validity of the signature provided in the response to the redeem script. Let `PubKeyA` and `PubKeyB` be public keys belonging to Alice and Bob respectively. In Script notation, the `<Check Sig>` portion is given below.

```
OP_IF
     OP_DROP <PubKeyB> OP_CHECKSIG
OP_ELSE
     OP_DROP <PubKeyA> OP_CHECKSIG
OP_ENDIF
```

The `OP_DROP` operator deletes the top stack element. It is used to get rid of the `SecretB` stack item as shown in Figure 6.7. We have assumed that the top stack element after the execution of the `<Compute Winner>` portion is 0.

For convenience, let us call the transaction created in this step the *funding transaction* as it funds the lottery by unlocking the UTXOs owned by Alice and Bob. It requires signatures from both Alice and Bob to be valid.

7. Alice includes her signature in the funding transaction and sends it to Bob. Bob includes his signature in the funding transaction and broadcasts it on the network. If Bob does not broadcast the funding transaction on the network, the lottery contract terminates. Alice and Bob reclaim their deposits by revealing their respective secret bytestrings. No one has lost any funds except for the transaction fees involved in recording the deposit transactions and the reclaim transactions on the blockchain.

[2] The `<Compute Winner>` script for deciding the winner is not unique and can be expressed in several other ways.

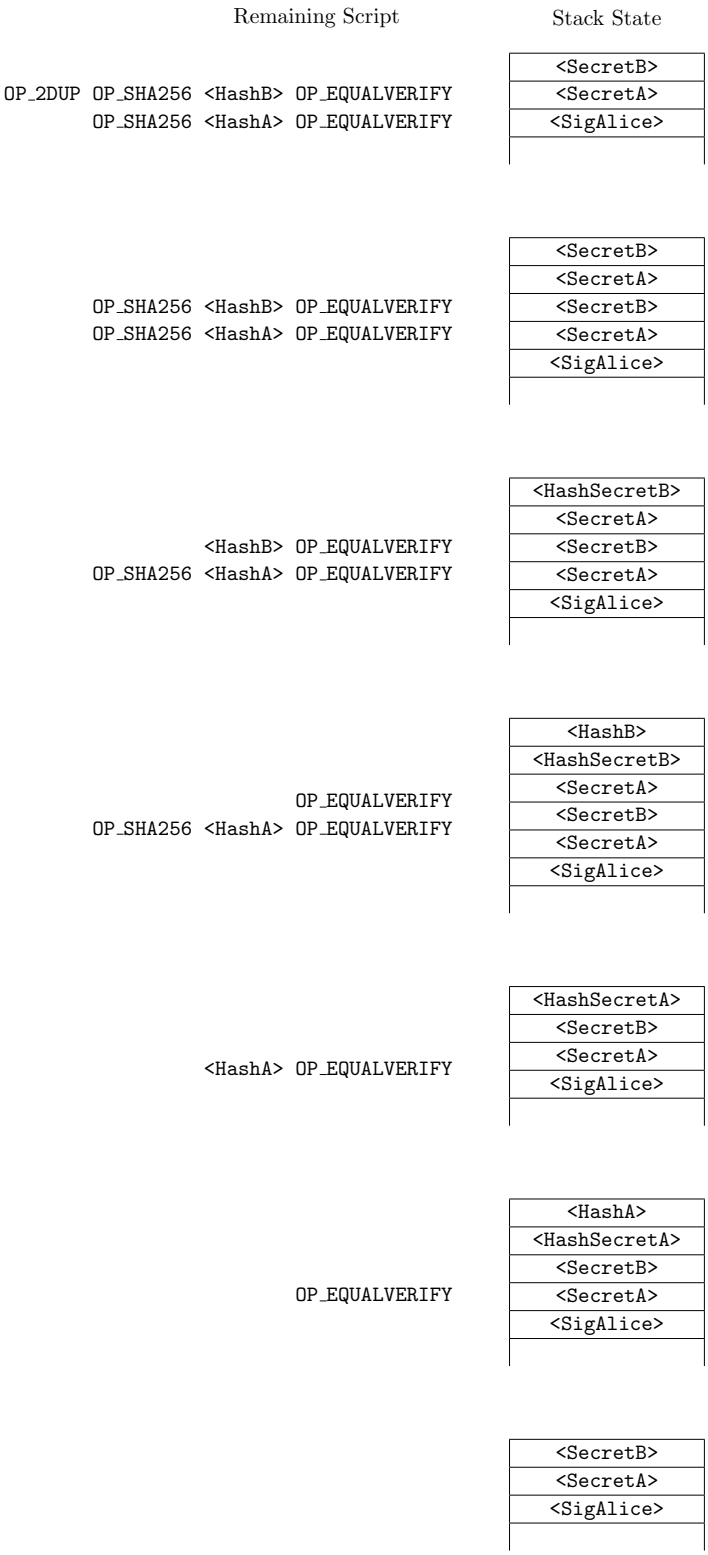


Figure 6.5: Stack state during the execution of the `<Check Hashes>` portion of the lottery redeem script

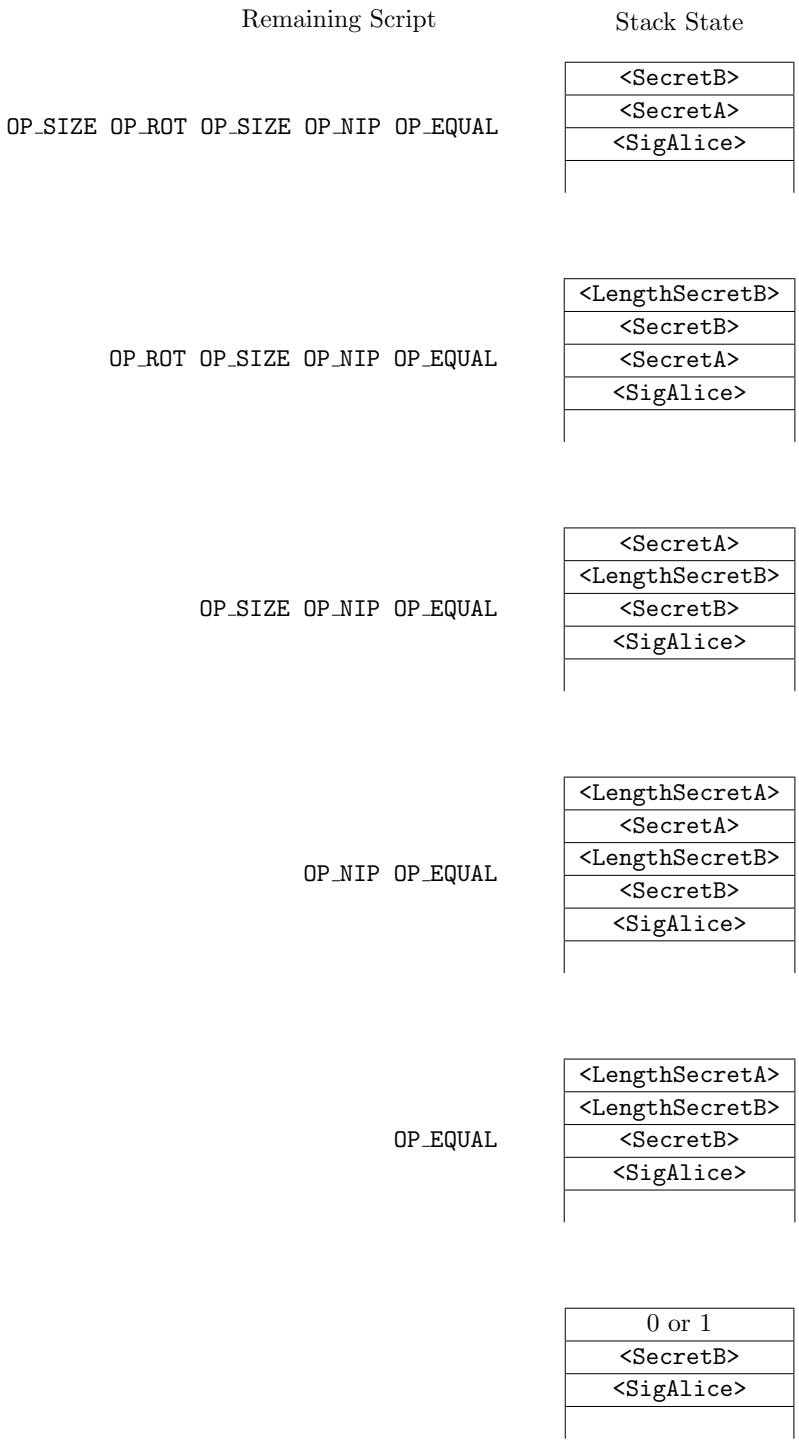


Figure 6.6: Stack state during the execution of the `<Compute Winner>` portion of the lottery redeem script

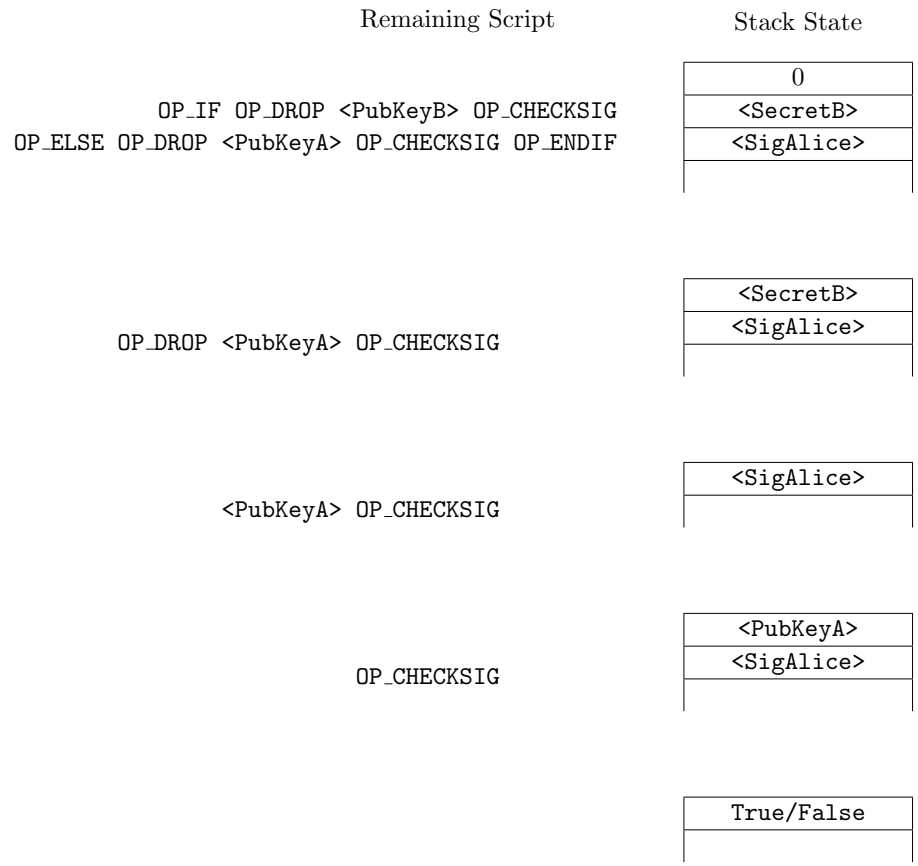


Figure 6.7: Stack state during the execution of the `<Check Sig>` portion of the lottery redeem script

8. If the funding transaction is broadcast and confirmed, Alice reveals her secret bytestring `SecretA` by claiming her deposit. Bob too claims his deposit and ends up revealing his secret `SecretB`. Now both Alice and Bob can compute the winner of the lottery by comparing the lengths of their secrets. If Alice is the winner, she broadcasts a transaction which unlocks the P2SH output and pays it to an address she owns. Bob does the same if he is the winner. In case either Alice or Bob decides to abort the contract by not revealing their secret bytestrings, the other party can claim the aborting party's deposit transaction output after `Timeout` is exceeded. Since the deposit transaction output holds two bitcoins (which is the same amount as the lottery jackpot), this situation is the same as the aborting party losing the lottery.

The generalization of the two-party lottery protocol to N parties involves each party betting one bitcoin and the winner (who can be any particular

party with probability $\frac{1}{N}$) receiving N bitcoins. The protocol proceeds as follows:

1. For $i = 0, 1, \ldots, N-1$, the ith party chooses a random integer l_i from the set $\{16, 17, \ldots, 16+N-1\}$ and generates a random bytestring s_i of length l_i bytes.

2. Both l_i and s_i are kept secret by each party. The parties exchange the hashes h_i of the secret bytestrings s_i.

3. For $i = 0, 1, \ldots, N-1$, the ith party broadcasts $N-1$ deposit transactions each of which has N bitcoins locked in its output. Each deposit transaction output can either be reclaimed by the ith party by revealing the secret s_i or by a jth party ($j \neq i$) after a timeout.

   If the ith party does not reveal s_i before the timeout, it will lose a total of $N(N-1)$ bitcoins which will distributed evenly among the other $N-1$ parties. So all other parties receive N bitcoins each. This ensures that all the honest parties are rewarded with the best possible outcome (winning the lottery) even if only one party aborts the protocol.

4. The winner of the lottery will be the party with index j where

$$j = \sum_{i=0}^{N-1} l_i \bmod N.$$

   All the parties sign a funding transaction which unlocks UTXOs containing one bitcoin owned by each of them and pays the sum to the lottery winner.

5. If all the parties reveal their secrets s_i before the timeout, the lottery winner claims the jackpot by spending the output in the funding transaction.

Chapter 7

Bitcoin Development

In this chapter, we describe the current process by which changes to the Bitcoin protocol are made. As mentioned before, the Bitcoin Core reference client implementation is the de facto specification of the Bitcoin protocol. The source code of this client is available in the Github repository at `https://github.com/bitcoin/bitcoin`.

7.1 Bitcoin Improvement Proposals

Any proposed change to the Bitcoin protocol begins with a Bitcoin Improvement Proposal (BIP). A BIP is a document with a precise technical description of the proposed change. Not all changes require a BIP. Minor code changes and enhancements can be directly submitted to the Bitcoin code repository at GitHub. They will be discussed on GitHub by the Bitcoin Core developers and be accepted or rejected once consensus has been reached.

For major changes, the person proposing the change discusses the proposed change on the bitcoin-dev mailing list.[1] This mailing list has all of the prominent Bitcoin Core developers as members. If there are no major objections regarding the feasibility or usefulness of the proposed change, the proposer posts a draft BIP on the bitcoin-dev mailing list. After the suggested changes have been incorporated, the draft BIP is submitted to the BIPs repository on GitHub available at `https://github.com/bitcoin/bips`. One of the Bitcoin Core developers is designated as the BIP editor (the current BIP editor is Luke Dashjr). The BIP editor assigns a unique BIP number to the draft BIP. Further changes can still be made to the draft BIP at this stage. The BIP editor also classifies the BIP as either a *process*, *informational*, or *standards track* BIP. The BIP category determines the subsequent BIP status changes.

- A process BIP describes a change to the Bitcoin development process. For example, the details of the BIP workflow are described in BIP 2

[1] `bitcoin-dev@lists.linuxfoundation.org`

which is titled "BIP Process, revised".[2] This BIP overrides BIP 1 which was the original specification of the BIP workflow. Process BIPs do not involve any code changes in the Bitcoin Core client. If there are no objections to a draft process BIP, its status changes to active.

- Informational BIPs either provide information to the Bitcoin community or describe features which do not affect block validity or the P2P network protocol. For example, BIP 50 contains a post-mortem report of an March 2013 fork in the blockchain due to unintentional differences in block validity rules between the 0.8 release of the Bitcoin Core client and the release prior to it.[3] Another example is BIP 173 which describes a new address format for SegWit outputs.[4] Like the P2PKH address format, this address format will be used for exchanging or representing addresses outside of the Bitcoin network. The blockchain will not store the outputs in this format. If the feature described in an informational BIP receives real-world adoption, its status is changed to final. Until such adoption becomes evident, its status may remain as draft or be changed to proposed once the BIP author deems the BIP to be complete.

- A standards track BIP describes a change which affects transaction/block validity rules or changes to the P2P network protocol. It may also describe changes which affect interoperability of different implementations of the Bitcoin protocol. Standards track BIPs must include a link to an implementation of the proposed feature along with the description of the feature design. Once the BIP proposer has a reference implementation of the draft BIP and does not anticipate any more changes, the status of the BIP is changed from draft to proposed. At this stage, the implementation related to the BIP can be merged into the Bitcoin Core code and made available in the next available release. If there is enough support for the BIP from the Bitcoin community, the BIP status is changed to final. The methodology used to measure support for a BIP is described in the next section.

Figure 7.1 illustrates the various BIP status transitions. The status of a draft BIP may be changed to deferred either by the BIP authors themselves at any time or by the BIP editor if there has not been any progress being made on the BIP. A deferred BIP may be changed back to a draft BIP once progress is made. A draft BIP may also be withdrawn by the BIP authors at any time. The status of a draft or proposed BIP is changed to rejected if there has been no progress for three years. The status of a final or active BIP is changed to replaced if another BIP supersedes the feature it describes. If

[2] `https://github.com/bitcoin/bips/blob/master/bip-0002.mediawiki`
[3] `https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki`
[4] `https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki`

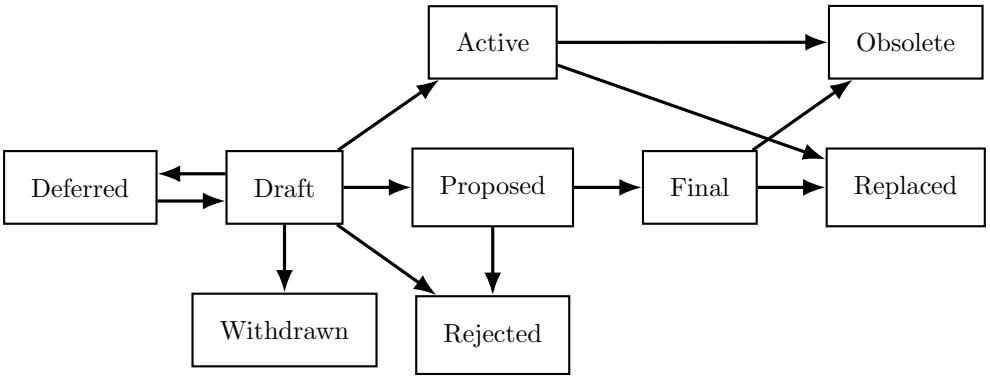


Figure 7.1: BIP status changes

the feature described in a final or active BIP is no longer relevant, its status is changed to obsolete.

7.2 Hard and Soft Forks

In Chapter 4, we discussed blockchain forks which occur when two miners find a valid block at the same time. Blockchain forks are eventually resolved as it is unlikely that both branches in the fork continue to be simultaneously extended to equal height. One of the branches is certain to overtake the other resulting in all the miners switching to it.

A blockchain fork can also occur due to changes to the Bitcoin protocol because all miners may not upgrade their Bitcoin clients to a version which includes the changes. *Hard forks* refer to protocol changes which require all the miners to upgrade to clients containing the changes in order to be successfully deployed. For example, suppose a change to the Bitcoin protocol raises the block size limit to 10 MB. For convenience, let us call the miners running old clients non-upgraded miners and the miners running upgraded clients upgraded miners. Now suppose an upgraded miner mines a 9 MB block and broadcasts it on the network. Non-upgraded miners will reject the block and continue working on the longest branch which contains only blocks which abide by the old rules. Upgraded miners will accept the block. This results in a blockchain fork with one branch containing the 9 MB block and the other branch containing only blocks which abide by the old rules. But this blockchain fork is seen only by the upgraded miners who consider both branches as valid. The non-upgraded miners see only the branch not containing the 9 MB block. The subsequent events can unfold in two ways.

- If the non-upgraded miners control the majority of the network hashrate, then the branch containing the 9 MB block will eventually be abandoned by the upgraded miners. This will happen for any branch containing

blocks which violate the old size limit. The net effect is that the deployment of the block size limit increase fails.

- If the upgraded miners control the majority of the network hashrate, the branch not containing the 9 MB block will be abandoned by them. But this branch will not be abandoned by the non-upgraded miners as it is the only valid branch they see. So as long non-upgraded miners exist, this branch will continue to be extended by them. For this reason, hard forks like block size limit increases require all the miners to upgrade for successful deployment.

Soft forks refer to protocol changes which require only miners controlling a majority of the network hashrate to upgrade in order to be successfully deployed. SegWit was a soft fork change to the Bitcoin protocol. For example, any P2WPKH output looks like an anyone-can-spend output to a miner running a pre-SegWit client. Such a miner will accept a transaction which spends this output by providing an empty `scriptSig` and no witness structure as valid. If a miner running a pre-SegWit client broadcasts a valid block containing the spending transaction, miners running pre-SegWit clients will accept the block while miners running SegWit clients will reject the block as invalid. The miners running SegWit clients will continue working on extending the longest branch that is valid under SegWit rules. Miners running pre-SegWit clients will see a blockchain fork with one branch containing the transaction spending the P2WPKH output without providing a witness structure and the other branch not containing this transaction. Miners running SegWit clients will not see the blockchain fork as they do not consider the transaction spending the P2WPKH output without providing a witness structure as valid. The subsequent events can unfold in two ways.

- If the miners running pre-SegWit clients control the majority of the network hashrate, then the branch not containing the transaction spending the P2WPKH output without providing a witness structure will be abandoned. This will prevent SegWit from being successfully deployed on the network.

- If the miners running SegWit clients control the majority of the network hashrate, then the branch containing the transaction spending the P2WPKH output without providing a witness structure will be abandoned by the miners running pre-SegWit clients. This will happen for any branch containing transactions which spend SegWit outputs without providing witness structures. Thus SegWit remains successfully deployed as long as miners controlling a majority of the network hashrate run SegWit clients.

To gauge miner readiness prior to activating a soft fork feature like SegWit, BIP 9 proposed allowing miners to indicate their readiness by setting bits in

the `nVersion` field of blocks they mine.[5] Each proposed soft fork is alloted a bit in the `nVersion` field. Recall that the mining target threshold is recalculated every 2016 blocks. This duration is called a retarget period. If at least 95% of the blocks ($\geq$ 1916 out of 2016) mined in a retarget period have the bit corresponding to a soft fork set, then the soft fork is considered locked-in and is activated at the end of the next retarget period.

[5] `https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki`

Appendix A

ECDSA Signature Malleability

Following the notation in Section 2.5, we show that $(r, n-s)$ is a valid `secp256k1` ECDSA signature for a message m whenever (r, s) is a valid `secp256k1` ECDSA signature for the same message.

Let e be the message digest (double SHA-256 hash) of m. Let $j \in \{1, 2, \ldots, n-1\}$ be a randomly chosen integer where n is the 256-bit prime number given in equation (2.3). Let $k \in \{1, 2, \ldots, n-1\}$ be the private key.

The signature generation procedure first adds the base point P to itself j times to get $jP = (x, y)$. The coordinates of the signature (r, s) are then $r = x \bmod n$ and $s = j^{-1}(e + kr) \bmod n$.

The signature verification procedure requires the signature (r, s), the message digest e, and the public key kP. The point $Q = j_1 P + j_2 kP$ is calculated where $j_1 = es^{-1} \bmod n$ and $j_2 = rs^{-1} \bmod n$. If $Q = (x_1, y_1) \in \mathbb{F}_p^2$, then the signature is considered valid if $r = x_1 \bmod n$.

We need the following lemma.

Lemma 1. *In the prime field $\mathbb{F}_n$, $(n-s)^{-1} = n - s^{-1}$ for all $s \in \mathbb{F}_n^*$.*

Proof. By definition, the multiplicative inverse of $n - s$ is $(n-s)^{-1}$. We see that $n - s^{-1}$ is also a multiplicative inverse of $n - s$ as

$$(n-s) * (n-s^{-1}) = (n^2 - sn - ns^{-1} + ss^{-1}) \bmod n = 1,$$
$$(n-s^{-1}) * (n-s) = (n^2 - s^{-1}n - ns + s^{-1}s) \bmod n = 1.$$

Every nonzero element in a field has a unique multiplicative inverse. To see this, suppose x and z are both multiplicative inverses of $y \in \mathbb{F}^*$, i.e. $x * y = y * x = 1$ and $y * z = z * y = 1$. Then $x = x*1 = x*(y*z) = (x*y)*z = 1*z = z$.

So by the uniqueness of the multiplicative inverse of $n - s$, we have $(n - s)^{-1} = n - s^{-1}$. □

Consider the verification procedure for the signature $(r, n-s)$ when (r, s) is a valid signature. The point Q corresponding to $(r, n-s)$ is given by

$$\begin{aligned}
Q &= e(n-s)^{-1}P + r(n-s)^{-1}kP \\
&= \left[(n-s)^{-1}(e+kr)\right]P \\
&\overset{(a)}{=} \left[(n-s^{-1})(e+kr)\right]P \\
&= nP - s^{-1}(e+kr)P \\
&\overset{(b)}{=} \mathcal{O} - s^{-1}(e+kr)P \\
&\overset{(c)}{=} -jP \\
&\overset{(d)}{=} (x, -y).
\end{aligned}$$

In the above equality chain, equality (a) follows from Lemma 1, equality (b) follows from $nP = \mathcal{O}$ which was argued in Section 2.4, equality (c) follows from $\mathcal{O}$ being the additive identity, and equality (d) follows from the fact that the additive identity of $jP = (x, y)$ is $(x, -y)$. Since $r = x \bmod n$, $(r, n-s)$ is a valid signature.

Appendix B

Probability of a successful double spending attack

In this appendix, we derive the success probability of a double spending attack given in equation (4.1). We closely follow the derivation by Meni Rosenfeld (`https://arxiv.org/abs/1402.2009v1`) with the exception that we do not assume that the attacker Alice pre-mines a block containing t_2 before commencing the attack.

To be completed.