Block Chain Technology

20CS4601C

UNIT_3

Cryptography and Technical Foundations: Cryptographic primitives, Asymmetric cryptography

Introducing Bitcoin: Overview, Cryptographic keys, transactions, Blockchain, Mining, Digital signatures, Wallets, Bitcoin improvement proposals (BIPs).

3
Symmetric Cryptography

In this chapter, you will be introduced to the concepts, theory, and practical aspects of *symmetric cryptography*. We will focus more on the elements that are specifically relevant in the context of blockchain technology. We will provide you with concepts that are required to understand the material covered in later chapters.

You will also be introduced to applications of cryptographic algorithms so that you can gain hands-on experience in the practical implementation of cryptographic functions. For this, we'll use the OpenSSL command-line tool. Before starting with the theoretical foundations, we'll look at the installation of OpenSSL in the following section, so that you can do some practical work as you read through the conceptual material.

Working with the OpenSSL command line

On the Ubuntu Linux distribution, OpenSSL is usually already available. However, it can be installed using the following command:

```
$ sudo apt-get install openssl
```

Examples in this chapter have been developed using OpenSSL version 1.0.2g.

It is available at `https://packages.ubuntu.com/xenial/openssl`.

You are encouraged to use this specific version, as all examples in the chapter have been developed and tested with it. The OpenSSL version can be checked using the following command:

```
$ openssl version
```

You will see the following output:

```
OpenSSL 1.0.2g  1 Mar 2016
```

Now, you are all set to run the examples provided in this chapter. If you are running a version other than 1.0.2g, the examples may still work but that is not guaranteed, as older versions lack the features used in the examples and newer versions may not be backward compatible with version 1.0.2g.

In the sections that follow, the theoretical foundations of cryptography are first discussed and then a series of relevant practical experiments will be presented.

Introduction

Cryptography is the science of making information secure in the presence of adversaries. It does so under the assumption that limitless resources are available to adversaries. **Ciphers** are algorithms used to encrypt or decrypt data, so that if intercepted by an adversary, the data is meaningless to them without **decryption**, which requires a secret key.

Cryptography is primarily used to provide a confidentiality service. On its own, it cannot be considered a complete solution, rather it serves as a crucial building block within a more extensive security system to address a security problem. For example, securing a blockchain ecosystem requires many different cryptographic primitives, such as hash functions, symmetric key cryptography, digital signatures, and public key cryptography.

In addition to a confidentiality service, cryptography also provides other security services such as integrity, authentication (entity authentication and data origin authentication), and non-repudiation. Additionally, accountability is also provided, which is a requirement in many security systems.

Before discussing cryptography further, some mathematical terms and concepts need to be explained in order to build a foundation for fully understanding the material provided later in this chapter.

The next section serves as a basic introduction to these concepts. An explanation with proofs and relevant background for all of these terms would require somewhat involved mathematics, which is beyond the scope of this book. More details on these topics can be found in any standard number theory, algebra, or cryptography-specific book. For example, *A Course in Number Theory and Cryptography* by *Neal Koblitz* provides an excellent presentation of all relevant mathematical concepts.

Mathematics

As the subject of cryptography is based on mathematics, this section will introduce some basic concepts that will help you understand the concepts presented later in the chapter.

Set

A **set** is a collection of distinct objects, for example, $X = \{1, 2, 3, 4, 5\}$.

Group

A **group** is a commutative set with one operation that combines two elements of the set. The group operation is closed and associated with a defined identity element. Additionally, each element in the set has an inverse. **Closure** (closed) means that if, for example, elements *A* and *B* are in the set, then the resultant element after performing an operation on the elements is also in the set. **Associative** means that the grouping of elements does not affect the result of the operation.

Field

A **field** is a set that contains both additive and multiplicative groups. More precisely, all elements in the set form an additive and multiplicative group. It satisfies specific axioms for addition and multiplication. For all group operations, the **distributive law** is also applied. The law dictates that the same sum or product will be produced even if any of the terms or factors are reordered.

A finite field

A **finite field** is one with a finite set of elements. Also known as *Galois fields*, these structures are of particular importance in cryptography as they can be used to produce accurate and error-free results of arithmetic operations. For example, prime finite fields are used in **Elliptic Curve Cryptography** (**ECC**) to construct discrete logarithm problems.

Order

The **order** is the number of elements in a field. It is also known as the *cardinality* of the field.

An abelian group

An **abelian group** is formed when the operation on the elements of a set is commutative. The commutative law means that changing the order of the elements does not affect the result of the operation, for example, $A \ X \ B = B \ X \ A$.

Prime fields

A **prime field** is a finite one with a prime number of elements. It has specific rules for addition and multiplication, and each nonzero element in the field has an inverse. Addition and multiplication operations are performed modulo *p*, that is, prime.

Ring

If more than one operation can be defined over an abelian group, that group becomes a **ring**. There are also specific properties that need to be satisfied. A ring must have closure and associative and distributive properties.

A cyclic group

A **cyclic group** is a type of group that can be generated by a single element called the *group generator*.

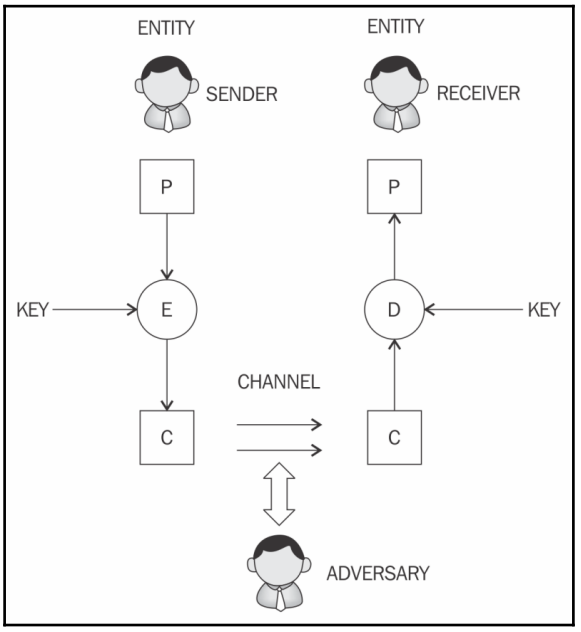
Modular arithmetic

Also known as clock arithmetic, numbers in modular arithmetic wrap around when they reach a certain fixed number. This fixed number is a positive number called **modulus**, and all operations are performed concerning this fixed number. Analogous to a clock, there are numbers from 1 to 12. When it reaches 12, the number 1 starts again. In other words, this type of arithmetic deals with the remainders after the division operation. For example, 50 mod 11 is 6 because 50 / 11 leaves a remainder of 6.

This completes a basic introduction to some mathematical concepts involved in cryptography. In the next section, you will be introduced to cryptography concepts.

Cryptography

A generic cryptography model is shown in the following diagram:



A model of the generic encryption and decryption model

In the preceding diagram, **P**, **E**, **C**, and **D** represent plaintext, encryption, ciphertext, and decryption, respectively. Also based on this model, explanations of concepts such as entity, sender, receiver, adversary, key, and channel follow:

- **Entity**: Either a person or system that sends, receives, or performs operations on data
- **Sender**: This is an entity that transmits the data
- **Receiver**: This is an entity that takes delivery of the data
- **Adversary**: This is an entity that tries to circumvent the security service
- **Key**: A key is data that is used to encrypt or decrypt other data
- **Channel**: Channel provides a medium of communication between entities

Next, we will describe the cryptography services mentioned earlier in the chapter in greater detail.

Confidentiality

Confidentiality is the assurance that information is only available to authorized entities.

Integrity

Integrity is the assurance that information is modifiable only by authorized entities.

Authentication

Authentication provides assurance about the identity of an entity or the validity of a message.

There are two types of authentication mechanisms, namely entity authentication and data origin authentication, which are discussed in the following section.

Entity authentication

Entity authentication is the assurance that an entity is currently involved and active in a communication session. Traditionally, users are issued a username and password that is used to gain access to the various platforms with which they are working. This practice is known as **single-factor authentication**, as there is only one factor involved, namely, *something you know*, that is, the password and username. This type of authentication is not very secure for a variety of reasons, for example, password leakage; therefore, additional factors are now commonly used to provide better security. The use of additional techniques for user identification is known as **multifactor authentication** (or two-factor authentication if only two methods are used).

Various authentication factors are described here:

- The first factor is *something you have*, such as a hardware token or a smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. This mechanism protects the user by requiring two factors of authentication. A user who has access to the hardware token and knows the login credentials will be able to access the system. Both factors should be available to gain access to the system, thus making this method a two-factor authentication mechanism. In case if the hardware token is lost, on its own it won't be of any use unless, *something you know*, the login password is also used in conjunction with the hardware token.
- The second factor is *something you are*, which uses biometric features to identify the user. With this method, a user's fingerprint, retina, iris, or hand geometry is used to provide an additional factor for authentication. This way, it can be ensured that the user was indeed present during the authentication process, as biometric features are unique to every individual. However, careful implementation is required to guarantee a high level of security, as some research has suggested that biometric systems can be circumvented under specific conditions.

Data origin authentication

Also known as *message authentication*, **data origin authentication** is an assurance that the source of the information is indeed verified. Data origin authentication guarantees data integrity because if a source is corroborated, then the data must not have been altered. Various methods, such as **Message Authentication Codes** (**MACs**) and digital signatures are most commonly used. These terms will be explained in detail later in the chapter.

Non-repudiation

Non-repudiation is the assurance that an entity cannot deny a previous commitment or action by providing incontrovertible evidence. It is a security service that offers definitive proof that a particular activity has occurred. This property is essential in debatable situations whereby an entity has denied the actions performed, for example, placement of an order on an e-commerce system. This service produces cryptographic evidence in electronic transactions so that in case of disputes, it can be used as a confirmation of an action.

Non-repudiation has been an active research area for many years. Disputes in electronic transactions are a common issue, and there is a need to address them to increase the confidence level of consumers in such services.

The non-repudiation protocol usually runs in a communication network, and it is used to provide evidence that an action has been taken by an entity (originator or recipient) on the network. In this context, there are two communications models that can be used to transfer messages from originator *A* to recipient *B*:

- A message is sent directly from originator *A* to recipient *B*.
- A message is sent to a delivery agent from originator *A*, which then delivers the message to recipient *B*.

The primary requirements of a non-repudiation protocol are fairness, effectiveness, and timeliness. In many scenarios, there are multiple participants involved in a transaction, as opposed to only two parties. For example, in electronic trading systems, there can be many entities, such as clearing agents, brokers, and traders that can be involved in a single transaction. In this case, two-party non-repudiation protocols are not appropriate. To address this problem, **Multi-Party Non-Repudiation** (**MPNR**) protocols have been developed.

Accountability

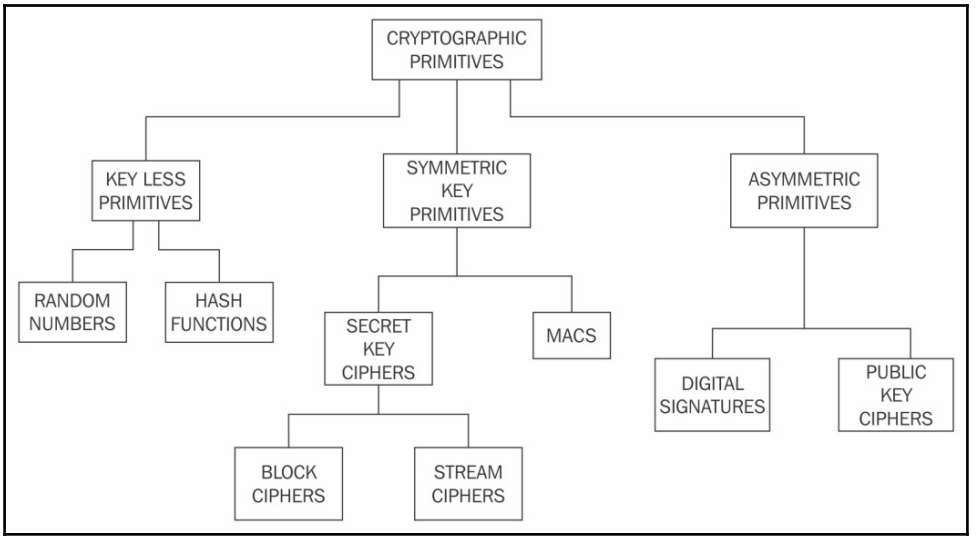
Accountability is the assurance which states that actions affecting security can be traced back to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are vital to trace an entity's actions, such as when a trade is placed in an audit record with the date and timestamp and the entity's identity is generated and saved in the log file. This log file can optionally be encrypted and be part of the database or a standalone ASCII text log file on a system.

In order to provide all of the services discussed earlier, different cryptographic primitives are used that are presented in the next section.

Cryptographic primitives

Cryptographic primitives are the basic building blocks of a security protocol or system. In the following section, you are introduced to cryptographic algorithms that are essential for building secure protocols and systems. A **security protocol** is a set of steps taken to achieve the required security goals by utilizing appropriate security mechanisms. Various types of security protocols are in use, such as authentication protocols, non-repudiation protocols, and key management protocols.

The taxonomy of cryptographic primitives can be visualized as shown here:



Cryptographic primitives

As shown in the cryptographic primitives taxonomy diagram, cryptography is mainly divided into two categories: *symmetric cryptography* and *asymmetric cryptography*.

These primitives are discussed further in the next section.

Symmetric cryptography

Symmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is the same one that is used for decrypting the data. Thus, it is also known as **shared key cryptography**. The key must be established or agreed upon before the data exchange occurs between the communicating parties. This is the reason it is also called **secret key cryptography**.

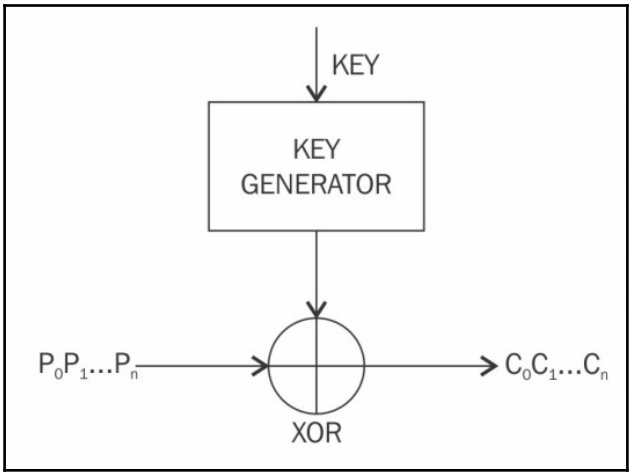
There are two types of symmetric ciphers: *stream ciphers* and *block ciphers*. **Data Encryption Standard** (**DES**) and **Advanced Encryption Standard** (**AES**) are typical examples of block ciphers, whereas RC4 and A5 are commonly used stream ciphers.

Stream ciphers

Stream ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis (one bit at a time) to plaintext using a keystream. There are two types of stream ciphers: *synchronous stream ciphers* and *asynchronous stream ciphers*:

- **Synchronous stream ciphers** are those where the keystream is dependent only on the key
- **Asynchronous stream ciphers** have a keystream that is also dependent on the encrypted data

In stream ciphers, encryption and decryption are the same function because they are simple modulo-2 additions or XOR operations. The fundamental requirement in stream ciphers is the security and randomness of keystreams. Various techniques ranging from pseudorandom number generators to true random number generators implemented in hardware have been developed to generate random numbers, and it is vital that all key generators be cryptographically secure:



Operation of a stream cipher

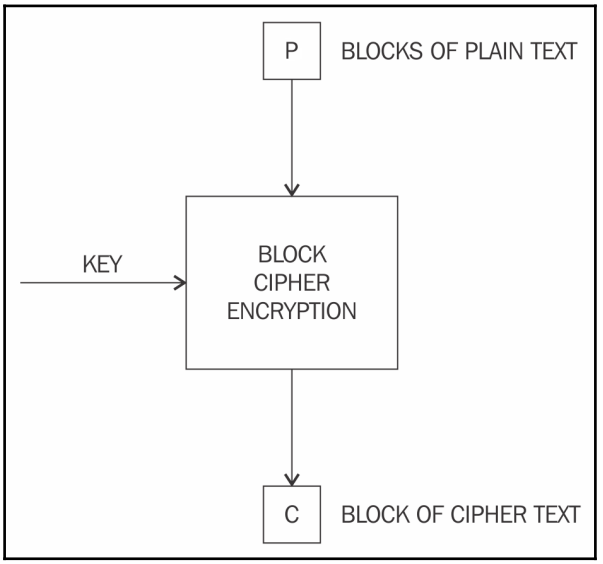
Block ciphers

Block ciphers are encryption algorithms that break up the text to be encrypted (plaintext) into blocks of a fixed length and apply the encryption block-by-block. Block ciphers are generally built using a design strategy known as a **Feistel cipher**. Recent block ciphers, such as AES (Rijndael) have been built using a combination of substitution and permutation called a **Substitution-Permutation Network** (**SPN**).

Feistel ciphers are based on the Feistel network, which is a structure developed by Horst Feistel. This structure is based on the idea of combining multiple rounds of repeated operations to achieve desirable cryptographic properties known as *confusion* and *diffusion*. Feistel networks operate by dividing data into two blocks (left and right) and processing these blocks via keyed *round functions* in iterations to provide sufficient pseudorandom permutation.

Confusion makes the relationship between the encrypted text and plaintext complex. This is achieved by substitution. In practice, *A* in plaintext is replaced by *X* in encrypted text. In modern cryptographic algorithms, substitution is performed using lookup tables called *S-boxes*. The diffusion property spreads the plaintext statistically over the encrypted data. This ensures that even if a single bit is changed in the input text, it results in changing at least half (on average) of the bits in the ciphertext. Confusion is required to make finding the encryption key very difficult, even if many encrypted and decrypted data pairs are created using the same key. In practice, this is achieved by transposition or permutation.

A key advantage of using a Feistel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process to achieve decryption. DES is a prime example of Feistel-based ciphers:



Simplified operation of a block cipher

Various modes of operation for block ciphers are **Electronic Code Book** (**ECB**), **Cipher Block Chaining** (**CBC**), **Output Feedback** (**OFB**) mode, and **Counter** (**CTR**) mode. These modes are used to specify the way in which an encryption function is applied to the plaintext. Some of these modes of block cipher encryption are introduced here.

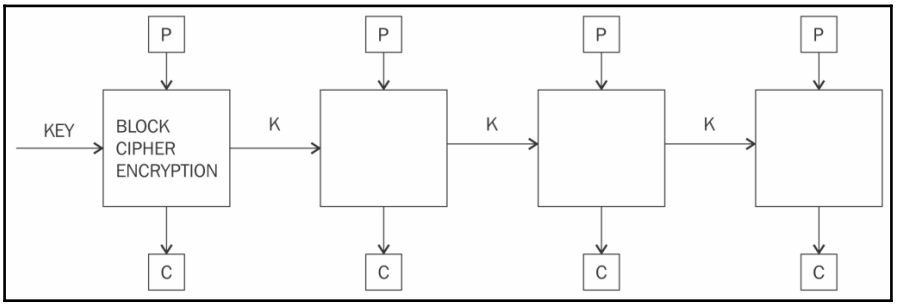
Block encryption mode

In **block encryption mode**, the plaintext is divided into blocks of fixed length depending on the type of cipher used. Then the encryption function is applied to each block.

The most common block encryption modes are briefly discussed in the following subsections.

Electronic Code Book

Electronic Code Book (**ECB**) is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm one-by-one to each block of plaintext. This is the most straightforward mode, but it should not be used in practice as it is insecure and can reveal information:

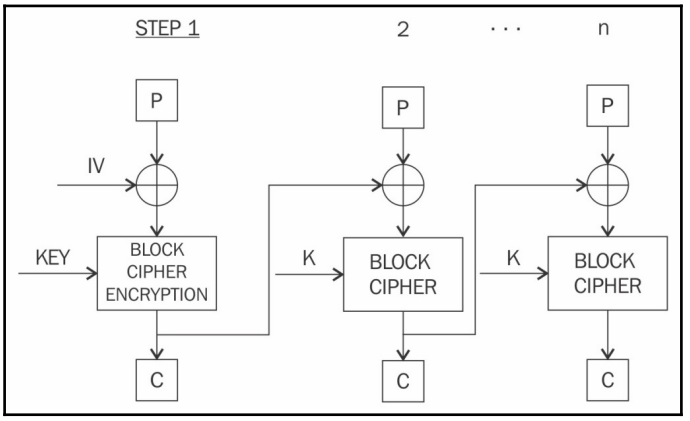


Electronic Code Book mode for block ciphers

The preceding diagram shows that we have plaintext **P** provided as an input to the block cipher encryption function, along with a key **KEY** and ciphertext **C** is produced as output.

Cipher Block Chaining

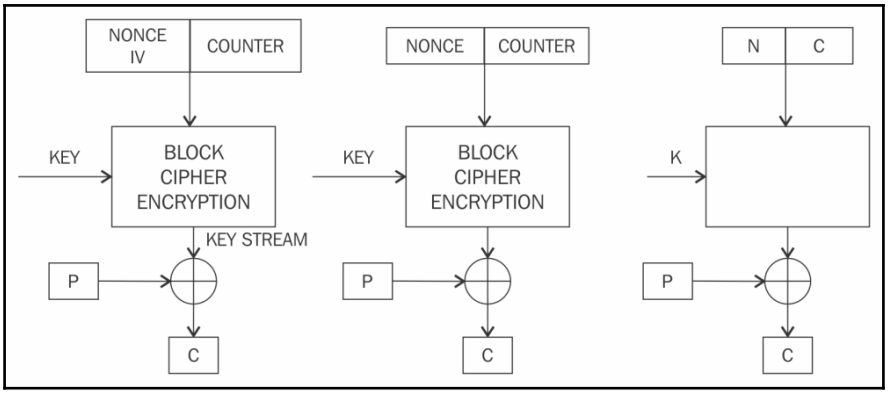
In **Cipher Block Chaining** (**CBC**) mode, each block of plaintext is XOR'd with the previously-encrypted block. CBC mode uses the **Initialization Vector** (**IV**) to encrypt the first block. It is recommended that the IV be randomly chosen:



Cipher block chaining mode

Counter mode

The **Counter** (**CTR**) mode effectively uses a block cipher as a stream cipher. In this case, a unique nonce is supplied that is concatenated with the counter value to produce a **keystream**:



Counter mode

There are other modes, such as **Cipher Feedback** (**CFB**) mode, **Galois Counte**r (**GCM**) mode, and **Output Feedback** (**OFB**) mode, which are also used in various scenarios.

Keystream generation mode

In **keystream generation mode**, the encryption function generates a keystream that is then XOR'd with the plaintext stream to achieve encryption.

Message authentication mode

In **message authentication mode**, a **Message Authentication Code** (**MAC**) results from an encryption function. The MAC is a cryptographic checksum that provides an integrity service. The most common method to generate a MAC using block ciphers is CBC-MAC, where a part of the last block of the chain is used as a MAC. For example, a MAC can be used to ensure that if a message is modified by an unauthorized entity. This can be achieved by encrypting the message with a key using the MAC function. The resultant message and MAC of the message once received by the receiver can be checked by encrypting the message received again by the key and comparing it with the MAC received from the sender. If they both match, then the message has not modified by unauthorized user thus integrity service is provided. If they both don't match, then it means that message is modified by unauthorized entity during the transmission.

Cryptographic hash mode

Hash functions are primarily used to compress a message to a fixed-length digest. In **cryptographic hash mode**, block ciphers are used as a compression function to produce a hash of plaintext.

With this, we have now concluded the introduction to block ciphers. In the following section, you will be introduced to the design and mechanism of a currently market-dominant block cipher known as AES.

Before discussing AES, however, some history is presented about the **Data Encryption Standard** (**DES**) that led to the development of the new AES standard.

Data Encryption Standard

The **Data Encryption Standard** (**DES**) was introduced by the U.S. **National Institute of Standards and Technology** (**NIST**) as a standard algorithm for encryption, and it was in widespread use during the 1980s and 1990s. However, it did not prove to be very resistant to brute force attacks, due to advances in technology and cryptography research. In July 1998, for example, the **Electronic Frontier Foundation** (**EFF**) broke DES using a special-purpose machine called EFF DES cracker (or *Deep Crack*).

DES uses a key of only 56 bits, which raised some concerns. This problem was addressed with the introduction of **Triple DES** (**3DES**), which proposed the use of a 168-bit key by means of three 56-bit keys and the same number of executions of the DES algorithm, thus making brute force attacks almost impossible. However, other limitations, such as slow performance and 64-bit block size, were not desirable.

Advanced Encryption Standard

In 2001, after an open competition, an encryption algorithm named Rijndael invented by cryptographers Joan Daemen and Vincent Rijmen was standardized as **Advanced Encryption Standard** (**AES**) with minor modifications by NIST. So far, no attack has been found against AES that is more effective than the brute-force method. The original version of Rijndael permits different key and block sizes of 128-bit, 192-bit, and 256-bits. In the AES standard, however, only a 128-bit block size is allowed. However, key sizes of 128-bit, 192-bit, and 256-bit are permissible.

How AES works

During AES algorithm processing, a 4 x 4 array of bytes known as the **state** is modified using multiple rounds. Full encryption requires 10 to 14 rounds, depending on the size of the key. The following table shows the key sizes and the required number of rounds:

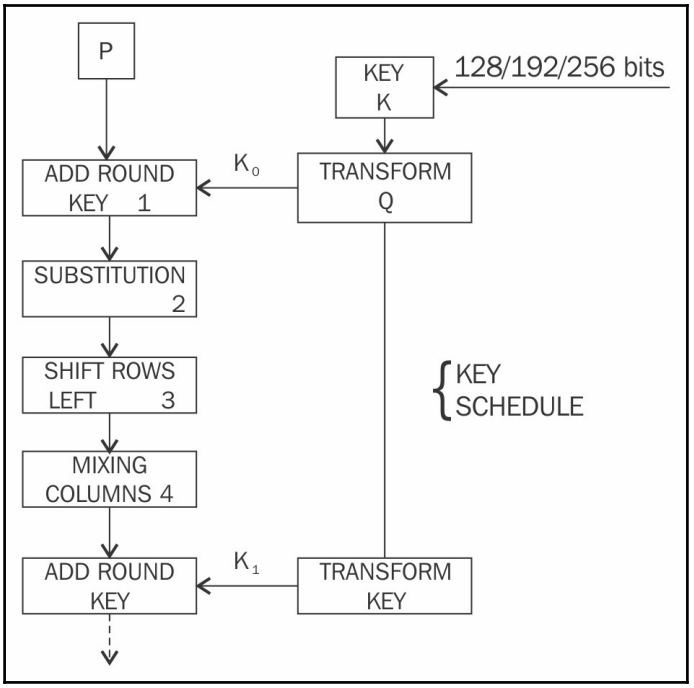
Key size	Number of rounds required
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

Once the state is initialized with the input to the cipher, four operations are performed in four stages to encrypt the input. These stages are: `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`:

1. In the `AddRoundKey` step, the state array is XOR'd with a subkey, which is derived from the master key
2. `SubBytes` is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array
3. The `ShiftRows` step is used to shift each row to the left, except for the first one, in the state array to the left in a cyclic and incremental manner
4. Finally, all bytes are mixed in the `MixColumns` step in a linear fashion, column-wise

The preceding steps describe one round of AES.

In the final round (either 10, 12, or 14, depending on the key size), stage 4 is replaced with `AddRoundKey` to ensure that the first three steps cannot be simply reversed:



AES block diagram, showing the first round of AES encryption. In the last round, the mixing step is not performed

Various cryptocurrency wallets use AES encryption to encrypt locally-stored data. Especially in Bitcoin wallet, AES-256 in the CBC mode is used.

Here's an OpenSSL example of how to encrypt and decrypt using AES:

```
$ openssl enc -aes-256-cbc -in message.txt -out message.bin
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
$ cat message.bin
```

The following are the contents of the `message.bin` file:

```
Salted__w    s_ÿ h~?                    :~/Crypt$
:~/Crypt$
```

Note that `message.bin` is a binary file. Sometimes, it is desirable to encode this binary file in a text format for compatibility/interoperability reasons. The following command can be used to do just that:

```
$ openssl enc -base64 -in message.bin -out message.b64
$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
$ cat message.b64
U2FsdGVkX193uByIcwZf0Z7J1at+4L+Fj8/uzeDAtJE=
```

In order to decrypt an AES-encrypted file, the following commands can be used. An example of `message.bin` from a previous example is used:

```
$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec
enter aes-256-cbc decryption password:
$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
$ cat message.dec
Datatoencrypt
```

Astute readers will have noticed that no IV has been provided, even though it's required in all block encryption modes of operation except ECB. The reason for this is that OpenSSL automatically derives the IV from the given password. Users can specify the IV using the following switch:

```
-K/-iv      , (Initialization Vector) should be provided in Hex.
```

In order to decode from base64, the following commands are used. Follow the `message.b64` file from the previous example:

```
$ openssl enc -d -base64 -in message.b64 -out message.ptx
$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 06:16 message.ptx
$ cat message.ptx
```

The following are the contents of the `message.ptx` file:

```
:~/Crypt$ cat message.ptx
Salted__w    s_ў  h~?                :~/Crypt$
```

There are many types of ciphers that are supported in OpenSSL. You can explore these options based on the preceding examples. A list of supported cipher types is shown in the following screenshot:

```
Cipher Types
-aes-128-cbc            -aes-128-ccm            -aes-128-cfb
-aes-128-cfb1           -aes-128-cfb8           -aes-128-ctr
-aes-128-ecb            -aes-128-ofb            -aes-192-cbc
-aes-192-ccm            -aes-192-cfb            -aes-192-cfb1
-aes-192-cfb8           -aes-192-ctr            -aes-192-ecb
-aes-192-ofb            -aes-256-cbc            -aes-256-ccm
-aes-256-cfb            -aes-256-cfb1           -aes-256-cfb8
-aes-256-ctr            -aes-256-ecb            -aes-256-ofb
-aes128                 -aes192                 -aes256
-bf                     -bf-cbc                 -bf-cfb
-bf-ecb                 -bf-ofb                 -blowfish
-camellia-128-cbc       -camellia-128-cfb       -camellia-128-cfb1
-camellia-128-cfb8      -camellia-128-ecb       -camellia-128-ofb
-camellia-192-cbc       -camellia-192-cfb       -camellia-192-cfb1
-camellia-192-cfb8      -camellia-192-ecb       -camellia-192-ofb
-camellia-256-cbc       -camellia-256-cfb       -camellia-256-cfb1
-camellia-256-cfb8      -camellia-256-ecb       -camellia-256-ofb
-camellia128            -camellia192            -camellia256
-cast                   -cast-cbc               -cast5-cbc
-cast5-cfb              -cast5-ecb              -cast5-ofb
-des                    -des-cbc                -des-cfb
-des-cfb1               -des-cfb8               -des-ecb
-des-ede                -des-ede-cbc            -des-ede-cfb
-des-ede-ofb            -des-ede3               -des-ede3-cbc
-des-ede3-cfb           -des-ede3-cfb1          -des-ede3-cfb8
-des-ede3-ofb           -des-ofb                -des3
-desx                   -desx-cbc               -id-aes128-CCM
-id-aes128-wrap         -id-aes192-CCM          -id-aes192-wrap
-id-aes256-CCM          -id-aes256-wrap         -id-smime-alg-CMS3DESwrap
-idea                   -idea-cbc               -idea-cfb
-idea-ecb               -idea-ofb               -rc2
-rc2-40-cbc             -rc2-64-cbc             -rc2-cbc
-rc2-cfb                -rc2-ecb                -rc2-ofb
-rc4                    -rc4-40                 -seed
-seed-cbc               -seed-cfb               -seed-ecb
-seed-ofb
```

Screenshot displaying rich library options available in OpenSSL

OpenSSL tool can be used to experiment with all the ciphers shown in the screenshot.

Summary

In this chapter, we introduced you to symmetric key cryptography. We started with basic mathematical definitions and cryptographic primitives. After this, we introduced you to the concepts of stream and block ciphers along with working modes of block ciphers. Moreover, we introduced you to the practical exercises using OpenSSL to complement the theoretical concepts.

In the next chapter, we will present public key cryptography, which is used extensively in blockchain technology and has very interesting properties.

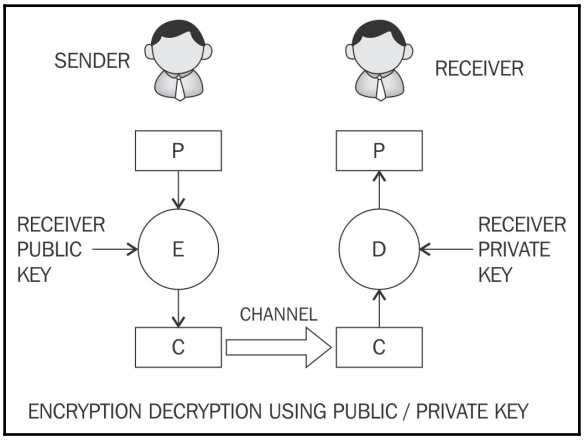
4
Public Key Cryptography

In this chapter, you will be introduced to the concepts and practical aspects of public key cryptography, also called asymmetric cryptography or asymmetric key cryptography. We will continue to use OpenSSL, as we did in the previous chapter, to experiment with some applications of cryptographic algorithms so that you can gain hands-on experience. We will start with the theoretical foundations of public key cryptography and will gradually build on the concepts with relevant practical exercises. In addition, we will also examine hash functions, which are another cryptographic primitive used extensively in blockchains. After this, we will introduce some new and advanced cryptography constructs.

Asymmetric cryptography

Asymmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is different from the key that is used to decrypt the data. This is also known as **public key cryptography**. It uses both public and private keys to encrypt and decrypt data, respectively. Various asymmetric cryptography schemes are in use, including RSA, DSA, and ElGammal.

An overview of public key cryptography is shown in the following diagram:

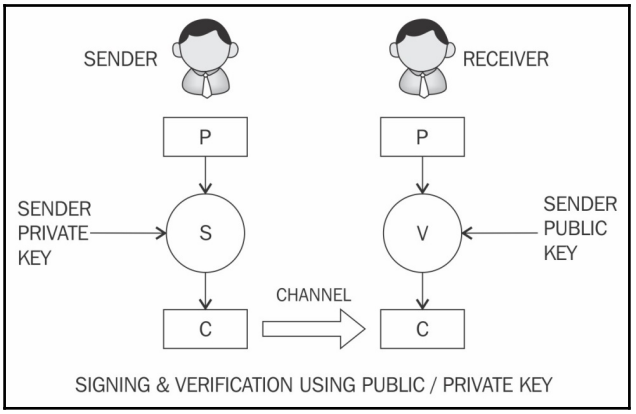


Encryption/decryption using public/private keys

The preceding diagram illustrates how a sender encrypts data **P** using the recipient's public key and encryption function **E** and producing an output encrypted data **C** which is then transmitted over the network to the receiver. Once it reaches the receiver, it can be decrypted using the receiver's private key by feeding the **C** encrypted data into function **D**, which will output plaintext **P**. This way, the private key remains on the receiver's side, and there is no need to share keys in order to perform encryption and decryption, which is the case with symmetric encryption.

The following diagram shows how the receiver uses public key cryptography to verify the integrity of the received message. In this model, the sender signs the data using their private key and transmits the message across to the receiver. Once the message is received, it is verified for integrity by the sender's public key.

It's worth noting that there is no encryption being performed in this model. It is simply presented here to help you understand thoroughly the sections covering message authentication and validation later in this chapter:



Model of a public-key cryptography signature scheme

The preceding diagram shows that sender digitally signs the plaintext **P** with his private key using signing function **S** and produces data **C** which is sent to the receiver who verifies **C** using sender public key and function **V** to ensure the message has indeed come from the sender.

Security mechanisms offered by public key cryptosystems include key establishment, digital signatures, identification, encryption, and decryption.

Key establishment mechanisms are concerned with the design of protocols that allow the setting up of keys over an insecure channel. Non-repudiation services, a very desirable property in many scenarios, can be provided using **digital signatures**. Sometimes, it is important not only to authenticate a user but also to identify the entity involved in a transaction. This can also be achieved by a combination of digital signatures and **challenge-response protocols**. Finally, the encryption mechanism to provide confidentiality can also be obtained using public key cryptosystems, such as RSA, ECC, and ElGammal.

Public key algorithms are slower in terms of computation than symmetric key algorithms. Therefore, they are not commonly used in the encryption of large files or the actual data that requires encryption. They are usually used to exchange keys for symmetric algorithm. Once the keys are established securely, symmetric key algorithms can be used to encrypt the data.

Public key cryptography algorithms are based on various underlying mathematical functions. The three main categories of asymmetric algorithms are described here.

Integer factorization

Integer factorization schemes are based on the fact that large integers are very hard to factor. RSA is the prime example of this type of algorithm.

Discrete logarithm

A **discrete logarithm scheme** is based on a problem in modular arithmetic. It is easy to calculate the result of modulo function, but it is computationally impractical to find the exponent of the generator. In other words, it is extremely difficult to find the input from the result. This is a one-way function.

For example, consider the following equation:

$$3^2 \bmod 10 = 9$$

Now, given *9*, the result of the preceding equation finding *2* which is the exponent of the generator *3* in the preceding question, is extremely hard to determine. This difficult problem is commonly used in the Diffie-Hellman key exchange and digital signature algorithms.

Elliptic curves

The **elliptic curves algorithm** is based on the discrete logarithm problem discussed earlier but in the context of elliptic curves. An **elliptic curve** is an algebraic cubic curve over a field, which can be defined by the following equation. The curve is non-singular, which means that it has no cusps or self-intersections. It has two variables *a* and *b*, as well as a point of infinity.

$$\boxed{y^2 = x^3 + ax + b}$$

Here, *a* and *b* are integers whose values are elements of the field on which the elliptic curve is defined. Elliptic curves can be defined over real numbers, rational numbers, complex numbers, or finite fields. For cryptographic purposes, an elliptic curve over prime finite fields is used instead of real numbers. Additionally, the prime should be greater than 3. Different curves can be generated by varying the value of *a* and/or *b*.

The most prominently used cryptosystems based on elliptic curves are the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) and the **Elliptic Curve Diffie-Hellman** (**ECDH**) key exchange.

To understand public key cryptography, the key concept that needs to be explored is the concept of public and private keys.

Public and private keys

A **private key**, as the name suggests, is a randomly generated number that is kept secret and held privately by its users. Private keys need to be protected and no unauthorized access should be granted to that key; otherwise, the whole scheme of public key cryptography is jeopardized, as this is the key that is used to decrypt messages. Private keys can be of various lengths depending on the type and class of algorithms used. For example, in RSA, typically a key of 1024-bits or 2048-bits is used. The 1024-bit key size is no longer considered secure, and at least a 2048-bit key size is recommended.

A **public key** is freely available and published by the private key owner. Anyone who would then like to send the publisher of the public key an encrypted message can do so by encrypting the message using the published public key and sending it to the holder of the private key. No one else is able to decrypt the message because the corresponding private key is held securely by the intended recipient. Once the public key encrypted message is received, the recipient can decrypt the message using the private key. There are a few concerns, however, regarding public keys. These include authenticity and identification of the publisher of the public keys.

In the following section, we will introduce two examples of asymmetric key cryptography: RSA and ECC. RSA is the first implementation of public key cryptography whereas ECC is used extensively in blockchain technology.

RSA

RSA was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adelman, hence the name **Rivest–Shamir–Adleman** (**RSA**). This type of public key cryptography is based on the integer factorization problem, where the multiplication of two large prime numbers is easy, but it is difficult to factor it (the result of multiplication, product) back to the two original numbers.

The crux of the work involved with the RSA algorithm is during the key generation process. An RSA key pair is generated by performing the following steps:

1. **Modulus generation**:

    - Select p and q, which are very large prime numbers
    - Multiply p and q, $n=p.q$ to generate modulus n

2. **Generate co-prime**:

    - Assume a number called e.
    - e should satisfy a certain condition; that is, it should be greater than 1 and less than $(p-1)(q-1)$. In other words, e must be a number such that no number other than 1 can divide e and $(p-1)(q-1)$. This is called **co-prime**, that is, e is the co-prime of $(p-1)(q-1)$.

3. **Generate the public key**:

    The modulus generated in step 1 and co-prime e generated in step 2 is a pair together that is a public key. This part is the public part that can be shared with anyone; however, p and q need to be kept secret.

4. **Generate the private key**:

    The private key, called d here, is calculated from p, q, and e. The private key is basically the inverse of e *modulo* $(p-1)(q-1)$. In the equation form, it is this as follows:

$$ed = 1 \bmod (p-1)(q-1)$$

Usually, the extended Euclidean algorithm is used to calculate d. This algorithm takes p, q, and e and calculates d. The key idea in this scheme is that anyone who knows p and q can easily calculate private key d by applying the extended Euclidean algorithm. However, someone who does not know the value of p and q cannot generate d. This also implies that p and q should be large enough for the modulus n to become extremely difficult (computationally impractical) to factor.

Encryption and decryption using RSA

RSA uses the following equation to produce ciphertext:

$$C = P^e \ mod \ n$$

This means that plaintext P is raised to e number of times and then reduced to modulo n. Decryption in RSA is provided in the following equation:

$$P = C^d \ mod \ n$$

This means that the receiver who has a public key pair (n, e) can decipher the data by raising C to the value of the private key d and reducing to modulo n.

Elliptic Curve Cryptography

Elliptic Curve Cryptography (**ECC**) is based on the discrete logarithm problem founded upon elliptic curves over finite fields (Galois fields). The main benefit of ECC over other types of public key algorithms is that it requires a smaller key size while providing the same level of security as, for example, RSA. Two notable schemes that originate from ECC are ECDH for key exchange and ECDSA for digital signatures.

ECC can also be used for encryption, but it is not usually used for this purpose in practice. Instead, it is used for key exchange and digital signatures commonly. As ECC needs less space to operate, it is becoming very popular on embedded platforms and in systems where storage resources are limited. By comparison, the same level of security can be achieved with ECC only using 256-bit operands as compared to 3072-bits in RSA.

Mathematics behind ECC

To understand ECC, a basic introduction to the underlying mathematics is necessary. An elliptic curve is basically a type of polynomial equation known as the **Weierstrass equation**, which generates a curve over a finite field. The most commonly-used field is where all the arithmetic operations are performed modulo a prime p. Elliptic curve groups consist of points on the curve over a finite field.

An elliptic curve is defined in the following equation:

$$y^2 = x^3 + Ax + B \bmod P$$

Here, *A* and *B* belong to a finite field *Zp* or *Fp* (prime finite field) along with a special value called the **point of infinity**. The point of infinity (∞) is used to provide identity operations for points on the curve.

Furthermore, a condition also needs to be met that ensures that the equation mentioned earlier has no repeated roots. This means that the curve is non-singular.

The condition is described in the following equation, which is a standard requirement that needs to be met. More precisely, this ensures that the curve is non-singular:

$$4a^3 + 27b^2 \neq 0 \bmod p$$

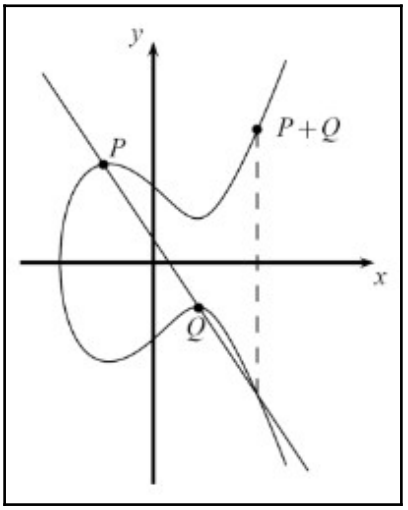
To construct the discrete logarithm problem based on elliptic curves, a large enough cyclic group is required. First, the group elements are identified as a set of points that satisfy the previous equation. After this, group operations need to be defined on these points.

Group operations on elliptic curves are point addition and point doubling. **Point addition** is a process where two different points are added, and **point doubling** means that the same point is added to itself.

Point addition

Point addition is shown in the following diagram. This is a geometric representation of point addition on elliptic curves. In this method, a diagonal line is drawn through the curve that intersects the curve at two points **P** and **Q**, as shown in the diagram, which yields a third point between the curve and the line. This point is mirrored as **P+Q**, which represents the result of the addition as **R**.

This is shown as **P+Q** in the following diagram:



Point addition over R

The group operation denoted by the + sign for addition yields the following equation:

$$P + Q = R$$

In this case, two points are added to compute the coordinates of the third point on the curve:

$$P + Q = R$$

More precisely, this means that coordinates are added, as shown in the following equation:

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

The equation of point addition is as follows:

$$X_3 = s^2 - x_1 - x_2 \bmod p$$

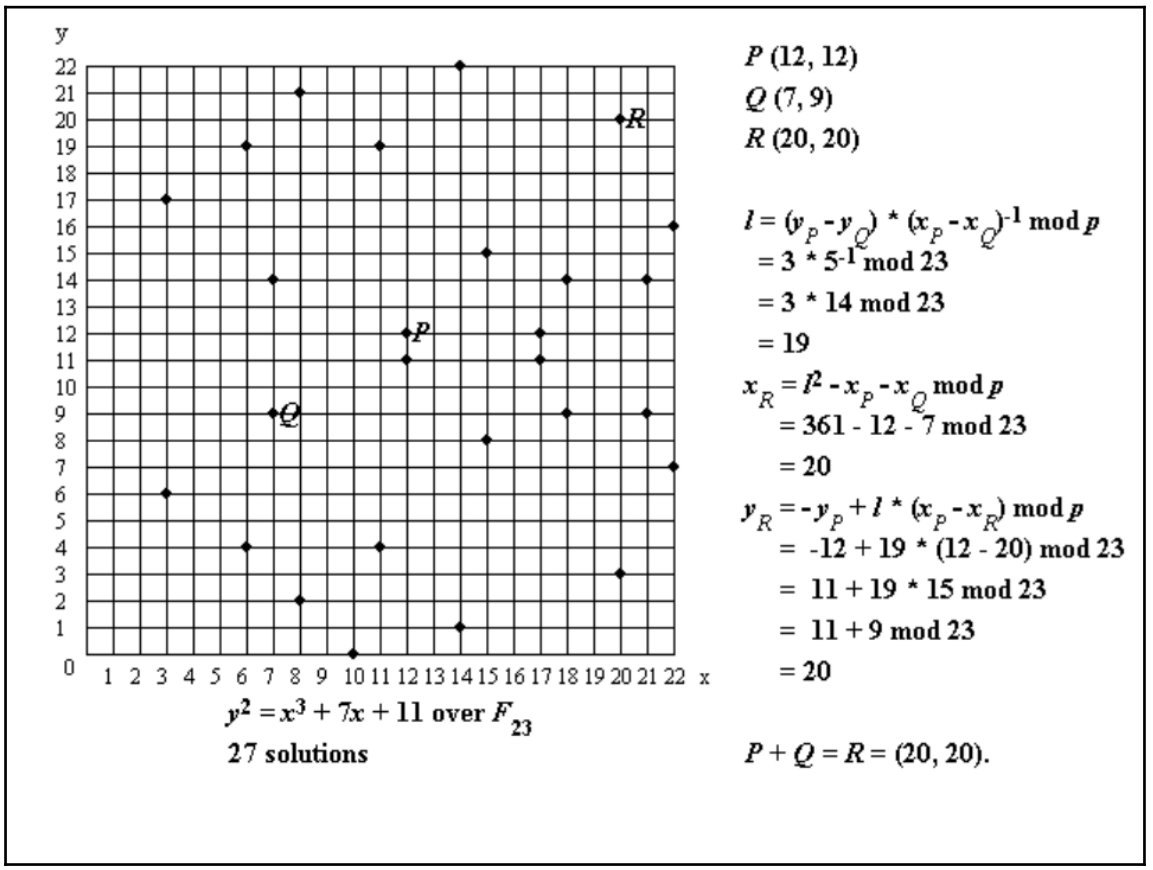
$$y_3 = s\,(x_1 - x_3) - y_1 \bmod p$$

Here, we see the result of the preceding equation:

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} mod\ p$$

S in the preceding equation depicts the line going through *P* and *Q*.

An example of point addition is shown in the following diagram. It was produced using Certicom's online calculator. This example shows the addition and solutions for the equation over finite field $\mathbf{F}_{23}$. This is in contrast to the example shown earlier, which is over real numbers and only shows the curve but provides no solutions to the equation:



Example of point addition

In the preceding example, the graph on the left side shows the points that satisfy this equation:

$$y^2 = x^3 + 7x + 11$$

There are 27 solutions to the equation shown earlier over finite field F_{23}. *P* and *Q* are chosen to be added to produce point *R*. Calculations are shown on the right side, which calculates the third point *R*. Note that here, *l* is used to depict the line going through *P* and *Q*.

As an example, to show how the equation is satisfied by the points shown in the graph, a point (*x*, *y*) is picked up where $x = 3$ and $y = 6$.

Using these values shows that the equation is indeed satisfied:

$$y^2 \ mod\ 23 = x^3 + 7x + 11 \ mod\ 23$$

$$6^2 \bmod 23 = 3^3 + 7(3) + 11 \ mod\ 23$$

$$36 \ mod\ 23 = 59 \ mod\ 23$$

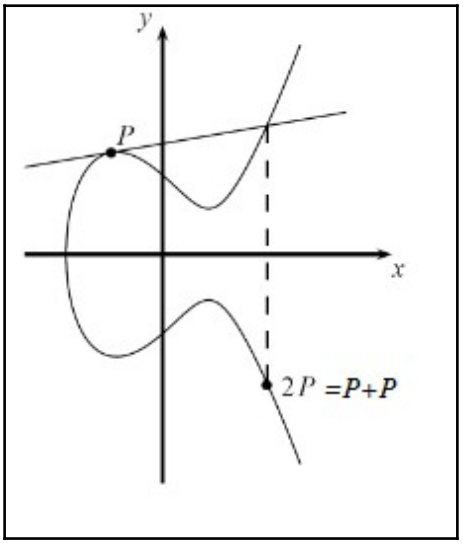
$$13 = 13$$

The next subsection introduces the concept of point doubling, which is another operation that can be performed on elliptic curves.

Point doubling

The other group operation on elliptic curves is called **point doubling**. This is a process where **P** is added to itself. In this method, a tangent line is drawn through the curve, as shown in the following graph. The second point is obtained, which is at the intersection of the tangent line drawn and the curve.

This point is then mirrored to yield the result, which is shown as *2P = P + P*:



Graph representing point doubling over real numbers

In the case of point doubling, the equation becomes:

$$x_3 = s^2 - x_1 - x_2 \ mod \ p$$

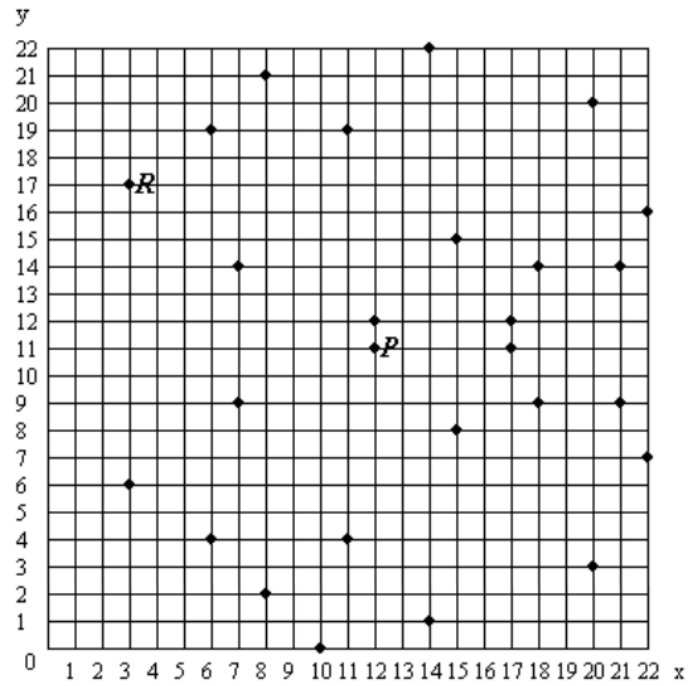
$$y_3 = s(x_1 - x_3) - y_1 \ mod \ p$$

$$S = \frac{3x_1^2 + a}{2y_1}$$

Here, *S* is the slope of tangent (tangent line) going through *P*. It is the line shown on top in the preceding diagram. In the preceding example, the curve is plotted over real numbers as a simple example, and no solution to the equation is shown.

The following example shows the solutions and point doubling of elliptic curves over finite field $\mathbf{F}_{23}$. The graph on the left side shows the points that satisfy the equation:

$$y^2 = x^3 + 7x + 11$$



P (12, 11)
R (3, 17)

$l = (3x_P^2 + a) * (2y_P)^{-1} \bmod p$
$= 439 * 22^{-1} \bmod 23$
$= 2 * 22 \bmod 23$
$= 21$

$x_R = l^2 - 2x_P \bmod p$
$= 441 - 24 \bmod 23$
$= 3$

$y_R = -y_P + l * (x_P - x_R) \bmod p$
$= -11 + 21 * (12 - 3) \bmod 23$
$= 12 + 21 * 9 \bmod 23$
$= 12 + 5 \bmod 23$
$= 17$

$2P = R = (3, 17)$.

Example of point doubling

As shown on the right side in the preceding graph, the calculation that finds the *R* after *P* is added into itself (point doubling). There is no *Q* as shown here, and the same point *P* is used for doubling. Note that in the calculation, *l* is used to depict the tangent line going through *P*.

In the next section, an introduction to the discrete logarithm problem will be presented.

Discrete logarithm problem in ECC

The discrete logarithm problem in ECC is based on the idea that, under certain conditions, all points on an elliptic curve form a cyclic group.

On an elliptic curve, the public key is a random multiple of the generator point, whereas the private key is a randomly chosen integer used to generate the multiple. In other words, a private key is a randomly selected integer, whereas the public key is a point on the curve. The discrete logarithm problem is used to find the private key (an integer) where that integer falls within all points on the elliptic curve. The following equation shows this concept more precisely.

Consider an elliptic curve *E*, with two elements *P* and *T*. The discrete logarithmic problem is to find the integer *d*, where *1 <= d <= #E*, such that:

$$\boxed{P + P + \cdots + P = d\,P = T}$$

Here, *T* is the public key (a point on the curve), and *d* is the private key. In other words, the public key is a random multiple of the generator, whereas the private key is the integer that is used to generate the multiple. *#E* represents the order of the elliptic curve, which means the number of points that are present in the cyclic group of the elliptic curve. A **cyclic group** is formed by a combination of points on the elliptic curve and point of infinity.

A key pair is linked with the specific domain parameters of an elliptic curve. Domain parameters include field size, field representation, two elements from the field *a* and *b*, two field elements *Xg* and *Yg*, order *n* of point *G* that is calculated as *G = (Xg, Yg)*, and the cofactor *h = #E(Fq)/n*. A practical example using OpenSSL will be described later in this section.

Various parameters are recommended and standardized to use as curves with ECC. An example of `secp256k1` specifications is shown here. This is the specification that is used in Bitcoin:

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve `secp256k1` are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F

$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

The curve E: $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

The base point G in compressed form is:

G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

and in uncompressed form is:

G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

Finally the order n of G and the cofactor are:

n = FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

h = 01

Specification of secp256k1 taken from http://www.secg.org/sec2-v2.pdf

An explanation of all of these values in the sextuple is as follows:

- *P* is the prime *p* that specifies the size of the finite field.
- *a* and *b* are the coefficients of the elliptic curve equation.
- *G* is the base point that generates the required subgroup, also known as the *generator*. The base point can be represented in either compressed or uncompressed form. There is no need to store all points on the curve in a practical implementation. The compressed generator works because the points on the curve can be identified using only the *x* coordinate and the least significant bit of the *y* coordinate.
- *n* is the order of the subgroup.
- *h* is the cofactor of the subgroup.

In the following section, two examples of using OpenSSL are shown to help you understand the practical aspects of RSA and ECC cryptography.

RSA using OpenSSL

The following example illustrates how RSA public and private key pairs can be generated using the OpenSSL command line.

RSA public and private key pair

First, how the RSA private key can be generated using OpenSSL is shown in the following subsection.

Private key

Execute the following command to generate the private key:

```
$ openssl genpkey -algorithm RSA -out privatekey.pem -pkeyopt \
 rsa_keygen_bits:1024
..............................++++++
..................++++++
```

The backslash (\) used in the commands are for continuation

After executing the command, a file named `privatekey.pem` is produced, which contains the generated private key as follows:

```
$ cat privatekey.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKJOFBzPy2vOd6em
Bk/UGrzDy7TvgDYnYxBfiEJId/r+EyMt/F14k2fDTOVwxXaXTxiQgD+BKuiey/69
9itnrqW/xy/pocDMvobj8QCngEntOdNoVSaN+t0f9nRM3iVM94mz3/C/v4vXvoac
PyPkr/0jhIV0woCurXGTghgqIbHRAgMBAAECgYEAlB3s/N4lJh0l1TkOSYunWtzT
6isnNkR7g1WrY9H+rG9xx4kP5b1DyE3SvxBLJA6xgBle8JVQMzm3sKJrJPFZzzT5
NNNnugCxairxcF1mPzJAP3aqpcSjxKpTv4qgqYevwgW1A0R3xKQZzBKU+bTO2hXV
D1oHxu75mDY3xCwqSAECQQDUYV04wNSEjEy9tYJ0zaryDAcvd/VG2/U/6qiQGajB
eSpSqoEESigbusKku+wVtRYgWWEomL/X58t+K01eMMZZAkEAw6PUR9YLebsm/Sji
iOShV4AKuFdi7t7DYWE5Ulb1uqP/i28zN/ytt4BXKIs/KcFykQGeAC6LDHZyycyc
ntDIOQJAVqrE1/wYvV5jkqcXbYLgV5YA+KYDOb9Y/ZRM5UETVKCVXNanf5CjfW1h
MMhfNxyGwvy2YVK0Nu8oY3xYPi+5QQJAUGcmORe4w6Cs12JUJ5p+zG0s+rG/URhw
B7djTXm7p6b6wR1EWYAZDM9MArenj8uXAA1AGCcIsmiDqHfU7lgz0QJAe9mOdNGW
7qRppgmOE5nuEbxkDSQI7OqHYbOLuwfCjHzJBrSgqyi6pj9/9CbXJrZPgNDwdLEb
GgpDKtZs9gLv3A==
-----END PRIVATE KEY-----
```

Public key

As the private key is mathematically linked to the public key, it is also possible to generate or derive the public key from the private key. Using the example of the preceding private key, the public key can be generated as shown here:

```
$ openssl rsa -pubout -in privatekey.pem -out publickey.pem

writing RSA key
```

The public key can be viewed using a file reader or any text viewer:

```
$ cat publickey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
74A2J2MQX4hCSHf6/hMjLfxdeJNnw0zlcMV2l08YkIA/gSronsv+vfYrZ66lv8cv
6aHAzL6G4/EAp4BJ7TnTaFUmjfrdH/Z0TN4lTPeJs9/wv7+L176GnD8j5K/9I4SF
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
```

In order to see more details of the various components, such as the modulus, prime numbers that are used in the encryption process, or exponents and coefficients of the generated private key, the following command can be used (only partial output is shown here as the actual output is very long):

```
$ openssl rsa -text -in privatekey.pem
Private-Key: (1024 bit)
modulus:
    00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
    1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
    77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
    c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
    f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
    f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
    f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
    be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
    ad:71:93:82:18:2a:21:b1:d1
publicExponent: 65537 (0x10001)
privateExponent:
    00:94:1d:ec:fc:de:25:26:1d:25:d5:39:0e:49:8b:
    a7:5a:dc:d3:ea:2b:27:36:44:7b:83:55:ab:63:d1:
    fe:ac:6f:71:c7:89:0f:e5:bd:43:c8:4d:d2:bf:10:
    4b:24:0e:b1:80:19:5e:f0:95:50:33:39:b7:b0:a2:
    6b:24:f1:59:cf:34:f9:34:d3:67:ba:00:b1:6a:2a:
    f1:70:5d:66:3f:32:40:3f:76:aa:a5:c4:a3:c4:aa:
    53:bf:8a:a0:a9:87:af:c2:05:b5:03:44:77:c4:a4:
    19:cc:12:94:f9:b4:ce:da:15:d5:0f:5a:07:c6:ee:
    f9:98:36:37:c4:2c:2a:48:01
prime1:
    00:d4:61:5d:38:c0:d4:84:8c:4c:bd:b5:82:74:cd:
    aa:f2:0c:07:2f:77:f5:46:db:f5:3f:ea:a8:90:19:
    a8:c1:79:2a:52:aa:81:04:4a:28:1b:ba:c2:a4:bb:
    ec:15:b5:16:20:59:61:28:98:bf:d7:e7:cb:7e:2b:
    4d:5e:30:c6:59
prime2:
    00:c3:a3:d4:47:d6:0b:79:bb:26:fd:28:e2:88:e4:
    a1:57:80:0a:b8:57:62:ee:de:c3:61:61:39:52:56:
    f5:ba:a3:ff:8b:6f:33:37:fc:ad:b7:80:57:28:8b:
    3f:29:c1:72:91:01:9e:00:2e:8b:0c:76:72:c9:cc:
    9c:9e:d0:c8:39
```

Exploring the public key

Similarly, the public key can be explored using the following commands. Public and private keys are base64-encoded:

```
$ openssl pkey -in publickey.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
74A2J2MQX4hCSHf6/hMjLfxdeJNnw0zlcMV2l08YkIA/gSronsv+vfYrZ66lv8cv
6aHAzL6G4/EAp4BJ7TnTaFUmjfrdH/Z0TN4lTPeJs9/wv7+L176GnD8j5K/9I4SF
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
Public-Key: (1024 bit)
Modulus:
   00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
   1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
   77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
   c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
   f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
   f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
   f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
   be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
   ad:71:93:82:18:2a:21:b1:d1
Exponent: 65537 (0x10001)
```

Now the public key can be shared openly, and anyone who wants to send you a message can use the public key to encrypt the message and send it to you. You can then use the corresponding private key to decrypt the file.

Encryption and decryption

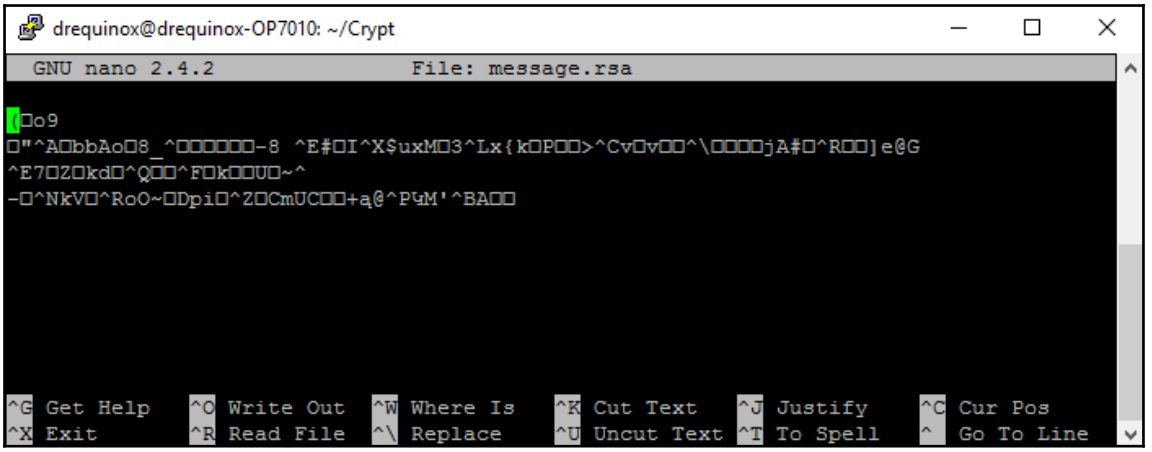
In this section, an example will be presented that demonstrates how encryption and decryption operations can be performed using RSA with OpenSSL.

Encryption

Taking the private key generated in the previous example, the command to encrypt a text file `message.txt` can be constructed as shown here:

```
$ echo datatoencrypt > message.txt
$ openssl rsautl -encrypt -inkey publickey.pem -pubin -in message.txt \
  -out message.rsa
```

This will produce a file named `message.rsa`, which is in a binary format. If you open `message.rsa` in the nano editor or any other text editor of your choice, it will show some garbage as shown in the following screenshot:



message.rsa showing garbage (encrypted) data

Decryption

In order to decrypt the RSA-encrypted file, the following command can be used:

```
$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa \
  -out message.dec
```

Now, if the file is read using `cat`, decrypted plaintext can be seen as shown here:

```
$ cat message.dec
datatoencrypt
```

ECC using OpenSSL

OpenSSL provides a very rich library of functions to perform ECC. The following subsection shows how to use ECC functions in a practical manner in OpenSSL.

ECC private and public key pair

In this subsection, first an example is presented that demonstrates the creation of a private key using ECC functions available in the OpenSSL library.

Private key

ECC is based on domain parameters defined by various standards. You can see the list of all available standards defined and recommended curves available in OpenSSL using the following command. (Once again, only partial output is shown here, and it is truncated in the middle.):

```
$ openssl ecparam -list_curves
secp112r1 : SECG/WTLS curve over a 112 bit prime field
secp112r2 : SECG curve over a 112 bit prime field
secp128r1 : SECG curve over a 128 bit prime field
secp128r2 : SECG curve over a 128 bit prime field
secp160k1 : SECG curve over a 160 bit prime field
secp160r1 : SECG curve over a 160 bit prime field
secp160r2 : SECG/WTLS curve over a 160 bit prime field
secp192k1 : SECG curve over a 192 bit prime field
secp224k1 : SECG curve over a 224 bit prime field
secp224r1 : NIST/SECG curve over a 224 bit prime field
secp256k1 : SECG curve over a 256 bit prime field
secp384r1 : NIST/SECG curve over a 384 bit prime field
secp521r1 : NIST/SECG curve over a 521 bit prime field
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
  .
  .
  .
  .
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field
brainpoolP384t1: RFC 5639 curve over a 384 bit prime field
brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
```

In the following example, `secp256k1` is employed to demonstrate ECC usage.

Private key generation

To generate the private key, execute the following command:

```
$ openssl ecparam -name secp256k1 -genkey -noout -out ec-privatekey.pem
$ cat ec-privatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIJHUIm9NZAgfpUrSxUk/iINq1ghM/ewn/RLNreuR52h/oAcGBSuBBAAK
oUQDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8JdcGXYGxHdzr0Jt1NInaYE0GG
ChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END EC PRIVATE KEY-----
```

The file named `ec-privatekey.pem` now contains the **Elliptic Curve** (**EC**) private key that is generated based on the `secp256k1` curve. In order to generate a public key from a private key, issue the following command:

```
$ openssl ec -in ec-privatekey.pem -pubout -out ec-pubkey.pem
read EC key
writing EC key
```

Reading the file produces the following output, displaying the generated public key:

```
$ cat ec-pubkey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8J
dcGXYGxHdzr0Jt1NInaYE0GGChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END PUBLIC KEY-----
```

Now the `ec-pubkey.pem` file contains the public key derived from `ec-privatekey.pem`. The private key can be further explored using the following command:

```
$ openssl ec -in ec-privatekey.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    00:91:d4:22:6f:4d:64:08:1f:a5:4a:d2:c5:49:3f:
    88:83:6a:d6:08:4c:fd:ec:27:fd:12:cd:ad:eb:91:
    e7:68:7f
pub:
    04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
    3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
    47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
    4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
    33:c1:62:a3:9e
ASN1 OID: secp256k1
```

Similarly, the public key can be further explored with the following command:

```
$ openssl ec -in ec-pubkey.pem -pubin -text -noout
read EC key
Private-Key: (256 bit)
pub:
    04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
    3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
    47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
    4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
    33:c1:62:a3:9e
ASN1 OID: secp256k1
```

It is also possible to generate a file with the required parameters, in this case, `secp256k1`, and then explore it further to understand the underlying parameters:

```
$ openssl ecparam -name secp256k1 -out secp256k1.pem
$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
```

The file now contains all the `secp256k1` parameters, and it can be analyzed using the following command:

```
$ openssl ecparam -in secp256k1.pem -text -param_enc explicit -noout
```

This command will produce the output similar to the one shown here:

```
Field Type: prime-field
Prime:
   00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
   ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
   ff:fc:2f
A:  0
B:  7 (0x7)
Generator (uncompressed):
   04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
   0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
   f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
   0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
   8f:fb:10:d4:b8
Order:
   00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
   ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
   36:41:41
Cofactor:  1 (0x1)
```

The preceding example shows the prime number used and values of `A` and `B`, with the generator, order, and cofactor of the `secp256k1` curve domain parameters.

With the preceding example, our introduction to public key cryptography from encryption and decryption perspective is complete. Other relevant constructs like digital signatures will be discussed later in the chapter.

In the next section, we will look at another category of cryptographic primitives, hash functions. Hash functions are not used to encrypt data; instead, they produce a fixed-length digest of the data that is provided as input to the hash function.

5
Introducing Bitcoin

Bitcoin is the first application of blockchain technology. In this chapter, you will be introduced to Bitcoin technology in detail.

Bitcoin has started a revolution with the introduction of the very first fully decentralized digital currency, and the one that has proven to be extremely secure and stable from a network and protocol point of view. As a currency bitcoin is quite unstable and highly volatile, albeit valuable. We will explain this later in the chapter. This has also sparked a great interest in academic and industrial research and introduced many new research areas.

Since its introduction in 2008 by Satoshi Nakamoto, Bitcoin has gained massive popularity, and it is currently the most successful digital currency in the world with billions of dollars invested in it. The current market cap, at the time of writing, for this currency is $149, 984, 293, 122. Its popularity is also evident from the high number of users and investors, increasing bitcoin price, everyday news related to Bitcoin, and the number of start-ups and companies that are offering bitcoin-based online exchanges, and it's now also traded as *Bitcoin Futures* on **Chicago Mercantile Exchange** (**CME**).

Interested readers can read more about *Bitcoin Futures* at `http://www.cmegroup.com/trading/bitcoin-futures.html`.

The name of the Bitcoin inventor *Satoshi Nakamoto* is believed to be a pseudonym, as the true identity of Bitcoin inventor is unknown. It is built on decades of research in the field of cryptography, digital cash, and distributed computing. In the following section, a brief history is presented in order to provide the background required to understand the foundations behind the invention of Bitcoin.

Digital currencies have always been an active area of research for many decades. Early proposals to create digital cash go as far back as the early 1980s. In 1982, David Chaum, a computer scientist, and cryptographer proposed a scheme that used blind signatures to build untraceable digital currency. This research was published in a research paper, *Blind Signatures for Untraceable Payments*.

Interested readers can read the original research paper which David Chaum describes his invention of the cryptographic primitive of blind signatures at `http://www.hit.bme.hu/~buttyan/courses/BMEVIHIM219/2009/Chaum.BlindSigForPayment.1982.PDF`.

In this scheme, a bank would issue digital money by signing a blind and random serial number presented to it by the user. The user could then use the digital token signed by the bank as currency. The limitation of this scheme was that the bank had to keep track of all used serial numbers. This was a central system by design and required to be trusted by the users.

Later on, in 1988, David Chaum and others proposed a refined version named e-cash that not only used a blinded signature, but also some private identification data to craft a message that was then sent to the bank.

Original research paper for this is available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.5759`.

This scheme allowed the detection of double spending but did not prevent it. If the same token was used at two different locations, then the identity of the double spender would be revealed. e-cash could only represent a fixed amount of money.

Adam Back, a cryptographer and now CEO of Blockstream, who is involved in blockchain development, introduced *hashcash* in 1997. It was originally proposed to thwart email spam. The idea behind hashcash was to solve a computational puzzle that was easy to verify but comparatively difficult to compute. The idea was that for a single user and a single email, the extra computational effort was negligible, but someone sending a large number of spam emails would be discouraged as the time and resources required to run the spam campaign would increase substantially.

In 1998, B-money was proposed by Wei Dai, a computer engineer who used to work for Microsoft, which introduced the idea of using **Proof of Work** (**PoW**) to create money. The term *Proof of Work* emerged and got popular later with Bitcoin, but in Wei Dai's B-money a scheme of creating money was introduced by providing a solution to a previously unsolved computational problem. It was referred in the paper as *solution to a previously unsolved computational problem*. This concept is similar to PoW, where money is created by broadcasting a solution to a previously unsolved computational problem.

The original paper is available at `http://www.weidai.com/bmoney.txt`.

A major weakness in the system was that an adversary with higher computational power could generate unsolicited money without allowing the network to adjust to an appropriate difficulty level. The system lacked details on the consensus mechanism between nodes and some security issues such as Sybil attacks were also not addressed. At the same time, Nick Szabo, a computer scientist introduced the concept of BitGold, which was also based on the PoW mechanism but had the same problems as B-money with the exception that the network difficulty level was adjustable. Tomas Sander and Amnon Ta-Shma from the **International Computer Science Institute** (**ICSI**), Berkley introduced an e-cash scheme under a research paper named *Auditable, Anonymous Electronic Cash* in 1999. This scheme, for the first time, used Merkle trees to represent coins and **Zero-Knowledge Proofs** (**ZKPs**) to prove the possession of coins.

The original research paper called *Auditable, Anonymous Electronic Cash* is available at: `http://www.cs.tau.ac.il/~amnon/Papers/ST.crypto99.pdf`.

In this scheme, a central bank was required that kept a record of all used serial numbers. This scheme allowed users to be fully anonymous. This was a theoretical design which was not practical to implement due to inefficient proof mechanisms.

Reusable Proof of Work (**RPoW**) was introduced in 2004 by Hal Finney, a computer scientist, developer and first person to receive Bitcoin from Satoshi Nakamoto. It used the hashcash scheme by Adam Back as a proof of computational resources spent to create the money. This was also a central system that kept a central database to keep track of all used PoW tokens. This was an online system that used remote attestation made possible by a trusted computing platform (TPM hardware).

All the previously mentioned schemes are intelligently designed but were weak from one aspect or another. Specifically, all these schemes rely on a central server that is required to be trusted by the users.

Bitcoin

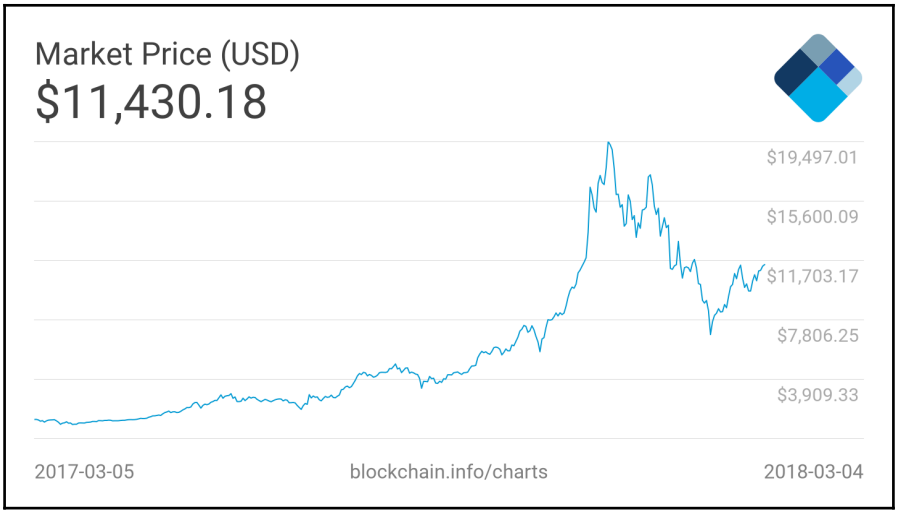
In 2008, Bitcoin was introduced through a paper called, *Bitcoin: A Peer-to-Peer Electronic Cash System*.

This paper is available at `https://bitcoin.org/bitcoin.pdf`.

It was written by Satoshi Nakamoto, which is believed to be a pseudonym, as the true identity of Bitcoin inventor is unknown and subject of much speculation. The first key idea introduced in the paper was of a purely peer-to-peer electronic cash that does need an intermediary bank to transfer payments between peers.

Bitcoin is built on decades of cryptographic research such as the research in Merkle trees, hash functions, public key cryptography, and digital signatures. Moreover, ideas such as BitGold, B-money, hashcash, and cryptographic time stamping provided the foundations for bitcoin invention. All these technologies are cleverly combined in Bitcoin to create the world's first decentralized currency. The key issue that has been addressed in Bitcoin is an elegant solution to the Byzantine Generals' Problem along with a practical solution of the double-spend problem. Recall, that both of these concepts are explained in `Chapter 1`, *Blockchain 101*.

The value of bitcoin has increased significantly since 2011, and then since March 2017 as shown in the following graph:



Bitcoin price since March 2017

The regulation of Bitcoin is a controversial subject and as much as it is a libertarian's dream, law enforcement agencies, governments and banks are proposing various regulations to control it, such as BitLicense issued by New York's state department of financial services. This is a license issued to businesses that perform activities related to virtual currencies. Due to high cost and very strict regulatory requirements pertaining to BitLicense many companies have withdrawn their services from New York.

For people with a libertarian ideology, Bitcoin is a platform which can be used instead of banks for business but they think that because of regulations, Bitcoin may become another institution which is not trusted. The original idea behind Bitcoin was to develop an e-cash system which requires no trusted third party and users can be anonymous. If regulations require **Know Your Customer** (**KYC**) checks and detailed information about business transactions to facilitate regulatory process then it might be too much information to share and as a result Bitcoin may not be attractive anymore to some.

There are now many initiatives being taken to regulate Bitcoin, cryptocurrencies and related activities such as ICOs. **Securities and Exchange Commission** (**SEC**) has recently announced that digital tokens, coins and relevant activities such as **Initial Coin Offerings** (**ICOs**) fall under the category of securities. This means that any digital currency trading platforms will need to be registered with SEC and will have all relevant securities laws and regulations applicable to them. This impacted the Bitcoin price directly and it fell almost 10% on the day this announcement was made.

Interested readers can read more about the regulation of Bitcoin and other relevant activities at `https://www.coindesk.com/category/regulation/`.

The growth of Bitcoin is also due to so-called **network effect**. Also called demand-side economies of scale, it is a concept that basically means more users who use the network, the more valuable it becomes. Over time, an exponential increase has been seen in the Bitcoin network growth. This increase in the number of users is largely financial gain driven. Also, the scarcity of Bitcoin and built-in inflation control mechanism gives it value as there are only 21 million bitcoins that can ever be mined and in addition the miner reward halves every four years. Even though the price of bitcoin fluctuates a lot, it has increased significantly over the last few years. Currently (at the time of writing this), bitcoin price is 9,250 U.S Dollars (USD).

Bitcoin definition

Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of peer-to-peer network, protocols, software that facilitate the creation and usage of the digital currency named bitcoin. Nodes in this peer-to-peer network talk to each other using the Bitcoin protocol.

Note that Bitcoin with a capital B is used to refer to the Bitcoin protocol, whereas bitcoin with a lowercase b is used to refer to bitcoin, the currency.

Decentralization of currency was made possible for the first time with the invention of bitcoin. Moreover, the double spending problem was solved in an elegant and ingenious way in bitcoin. Double spending problem arises when, for example, a user sends coins to two different users at the same time and they are verified independently as valid transactions. The double spending problem is resolved in Bitcoin by using a distributed ledger (blockchain) where every transaction is recorded permanently and by implementing transaction validation and confirmation mechanism. This process will be explained later in the chapter where we introduce the concept of *mining*.

Bitcoin – a bird's-eye view

In this section, we will see how the Bitcoin network looks from a user's point of view. How a transaction is made, how it propagates from a user to the network, how transactions are verified, and finally accumulated in blocks. We will look at what are the various actors and components of the Bitcoin network. Finally, some discussion on how all actors and components interact with each other to form the Bitcoin network will also be provided.

First, let us see that what the main components of a Bitcoin network are. Bitcoin is composed of the elements listed here. We will further expand on these elements as we progress through the chapter.

- Digital keys
- Addresses
- Transactions
- Blockchain
- Miners
- The Bitcoin network
- Wallets (client software)

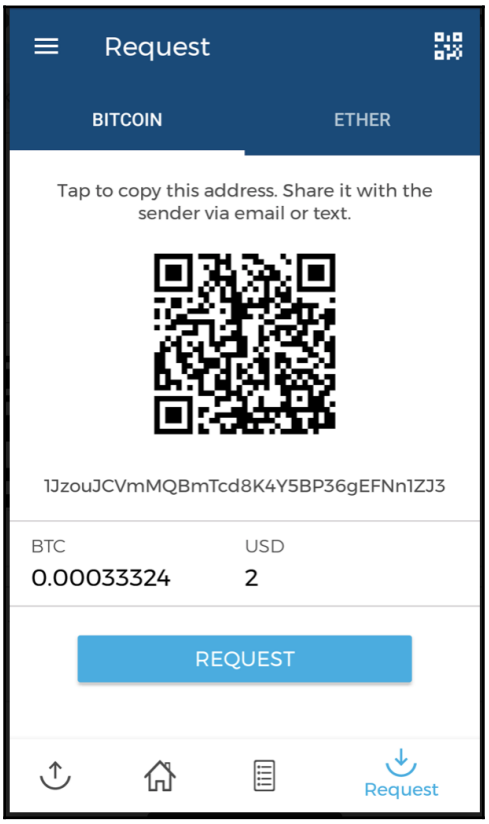
Now, we will see that how a user will use the Bitcoin network. The following example will help you understand that how the Bitcoin network looks like from an end user's perspective. We will see that what actors and components are involved in a Bitcoin transaction. One of the most common transactions is sending money to someone else, therefore in the following example we will see that how a payment can be sent from one user to another on the Bitcoin network.

Sending a payment to someone

This example will demonstrate that how money can be sent using Bitcoin network from one user to another. There are several steps that are involved in this process. As an example, we are using Blockchain wallet for mobile devices.

The steps are described here:

1. First, either the payment is requested from a user by sending his Bitcoin address to the sender via email or some other means such as SMS, chat applications or in fact any appropriate communication mechanism. The sender can also initiate a transfer to send money to another user. In both cases, the address of beneficiary is required. As an example, the Blockchain wallet is shown here where a payment request is created:



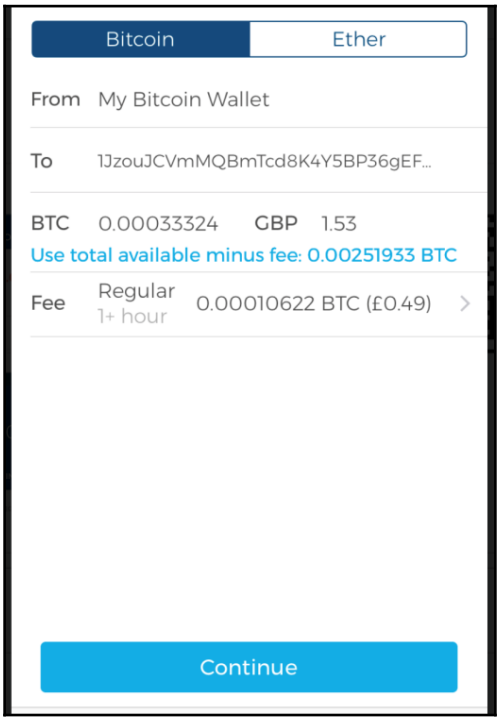
bitcoin payment request (using Blockchain wallet)

2. The sender either enters the receiver's address or scans the QR code that has the Bitcoin address, amount and optional description encoded in it. The wallet application recognizes this QR code and decodes it into something like `Please send <Amount> BTC to the Bitcoin address <receiver's Bitcoin address>`.
3. This will look like as shown here with values: `Please send 0.00033324 BTC to the Bitcoin address 1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3`.
4. This is also shown in the screenshot presented here:

Bitcoin payment QR code

The QR code shown in the preceding screenshot is decoded to `bitcoin://1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3?amount=0.00033324` which can be opened as a URL in Bitcoin wallet.

5. In the wallet application of the sender, this transaction is constructed by following some rules and broadcasted to the Bitcoin network. This transaction is digitally signed using the private key of the sender before broadcasting it. How the transaction is created, digitally signed, broadcasted, validated and added to the block will become clear in the following sections. From a user's point of view, once the QR code is decoded the transaction will appear similar to what is shown in the following screenshot:

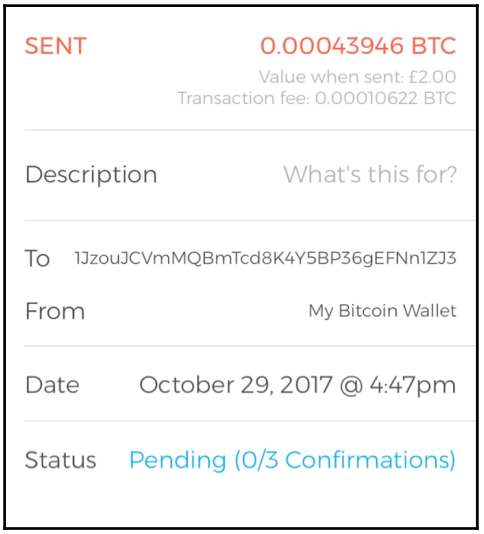


Send BTC using Blockchain wallet

Note that in the preceding screenshot there are a number of fields such as **From**, **To**, **BTC**, and **Fee**. While other fields are self-explanatory, it's worth noting that **Fee** is calculated based on the size of the transaction and a fee rate is a value that depends on the volume of the transaction in the network. This is represented in Satoshis/byte. Fee in Bitcoin network ensures that your transaction will be included by miners in the block.

Recently the Bitcoin fees were so high that even for smaller transactions a high amount of fee was charged. This was due to the fact that miners are free to choose which transactions they pick to verify and add in a block, and they select the ones with higher fees. The high number of users creating thousands of transactions also played a role in causing this situation of high fees because transactions were competing with each other to be picked up first and miners picked up the ones with highest fees. This fee is also usually estimated and calculated by the Bitcoin wallet software automatically before sending the transactions. The higher the transaction fee the more chances are that your transaction will be picked up at priority and included in the block. This task is performed by the miners. Mining and miners is a concept that we will look at a bit later in this chapter in the context of Bitcoin mining.

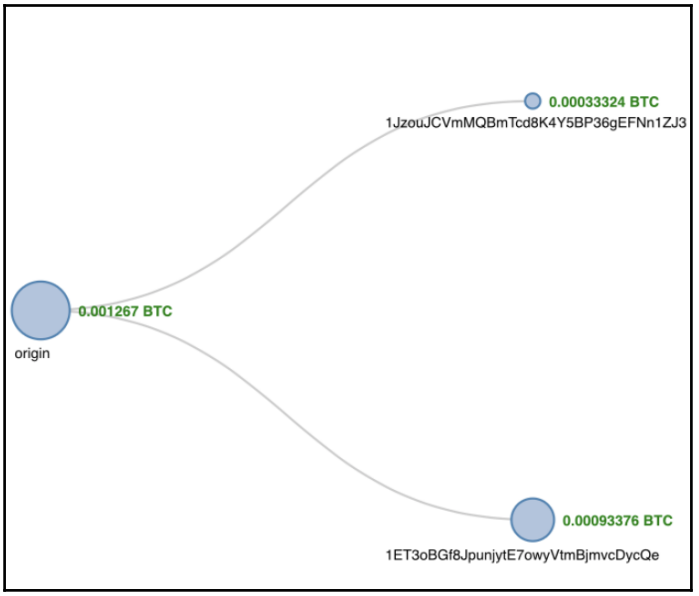
Once the transaction is sent it will appear as shown here in the Blockchain wallet software:



Transaction sent

4. At this stage, the transaction has been constructed, signed and sent out to the Bitcoin network. This transaction will be picked up by miners to be verified and included in the block. Also note that in the preceding screenshot, confirmation is pending for this transaction. These confirmations will start to appear as soon as the transaction is verified, included in the block, and mined. Also, the appropriate fee will be deducted from the original value to be transferred and will be paid to the miner who has included it in the block for mining.

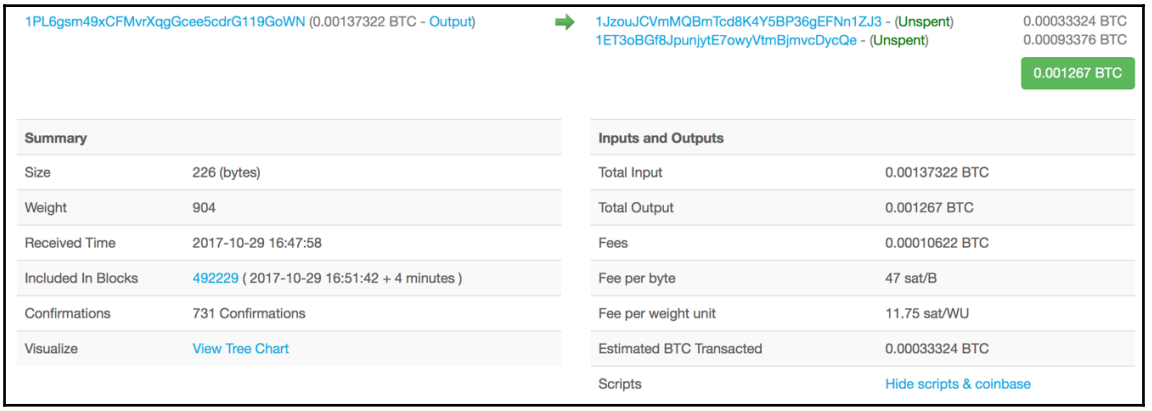
This flow is shown in the following diagram, where a payment of 0.001267 BTC (approximately 11 USD) is originated from the sender's address and been paid to receiver's address (starting with 1Jz). The fee of 0.00010622 (approximately 95 cents) is also deducted from the transaction as mining fee.



Transaction flow visualization (Blockchain.info)

The preceding screenshot visually shows how the transaction flowed on the network from origin (sender) to receivers on the right-hand side.

A summary view of various attributes of the transaction is shown here:



Snapshot of the transaction taken from Blockchain.info

Looking at the preceding screenshot there are a number of fields that contain various values. Important fields are listed here with their purpose and explanation:

- **Size**: This is the size of the transaction in bytes.
- **Weight**: This is the new metric given for block and transaction size since the introduction of **Segregated Witness** (**SegWit**) version of Bitcoin.
- **Received Time**: This is the time when the transaction is received.
- **Included In Blocks**: This shows the block number on the blockchain in which the transaction is included.
- **Confirmations**: This is the number of confirmations by miners for this transaction.
- **Total Input**: This is the number of total inputs in the transaction.
- **Total Output**: This is the number of total outputs in the transaction.
- **Fees**: This is the total fees charged.
- **Fee per byte**: This field represents the total fee divided by the number of bytes in a transaction. For example 10 Satoshis per byte.
- **Fee per weight unit**: For legacy transaction it is calculated using *total number of bytes * 4*. For SegWit transactions it is calculated by combining SegWit marker, flag, and witness field as one weight unit and each byte of other fields as four weight units.

Transaction ID of this transaction on the Bitcoin network is `d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d` and can be further explored using the `https://blockchain.info/tx/d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d` link via services provided by `https://blockchain.info/`. This transaction ID is available in the wallet software after transaction is sent to the network. From there it can be further explored using one of many Bitcoin blockchain explorers available online. We are using `https://blockchain.info/` as an example.

Bitcoin transactions are serialized for transmission over the network and encoded in hexadecimal format. As an example, the preceding transaction, is also shown here. We will see later in the *Transactions* section that how this hexadecimal encoded transaction can be decoded and what fields make up a transaction.

```
01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef90ff0430e12408
b0000006b483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7afb
f92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c717
58d012102c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12ff
ffffff022c820000000000001976a914c568ffeb46c6a9362e44a5a49deaa6eab05a619a88a
cc06c0100000000001976a9149386c8c880488e80a6ce8f186f788f3585f74aee88ac000000
00
```

In summary, the payment transaction in the Bitcoin network can be divided into the following steps:

1. Transaction starts with a sender signing the transaction with their private key
2. Transaction is serialized so that it can be transmitted over the network
3. Transaction is broadcasted to the network
4. Miners listening for the transactions picks up the transaction
5. Transaction are verified for their validity by the miners
6. Transaction are added to the candidate/proposed block for mining
7. Once mined, the result is broadcasted to all nodes on the Bitcoin network

Mining, transaction and other relevant concepts will become clearer in the following sections in the chapter. Now in the next section various denominations of bitcoin are presented.

The bitcoin currency, being digital has various denominations which are shown in the following table. A sender or receiver can request any amount. The smallest bitcoin denomination is the Satoshi. The bitcoin currency units are described as follows:

DENOMINATION	ABBREVIATION	FAMILIAR NAME	VALUE IN BTC
Satoshi	SAT	Satoshi	0.00000001 BTC
Microbit	µBTC (uBTC)	Microbitcoin or Bit	0.000001 BTC
Millibit	mBTC	Millibitcoin	0.001 BTC
Centibit	cBTC	Centibitcoin	0.01 BTC
Decibit	dBTC	Decibitcoin	0.1 BTC
Bitcoin	BTC	Bitcoin	1 BTC
DecaBit	daBTC	Decabitcoin	10 BTC
Hectobit	hBTC	Hectobitcoin	100 BTC
Kilobit	kBTC	Kilobitcoin	1000 BTC
Megabit	MBTC	Megabitcoin	1000000 BTC

bitcoin denominations

Now you will be introduced to the building blocks of Bitcoin one by one. First, we will look at the keys and addresses which are used to represent the ownership and transfer of value on the Bitcoin network.

Digital keys and addresses

On the Bitcoin network, possession of bitcoins and transfer of value via transactions is reliant upon private keys, public keys, and addresses. In `Chapter 4`, *Public Key Cryptography*, we have already covered these concepts, and here we will see that how private and public keys are used in the Bitcoin network.

Elliptic Curve Cryptography (**ECC**) is used to generate public and private key pairs in the Bitcoin network.

Private keys in Bitcoin

Private keys are required to be kept safe and normally resides only on the owner's side. Private keys are used to digitally sign the transactions proving the ownership of the bitcoins.

Private keys are fundamentally 256-bit numbers randomly chosen in the range specified by the `secp256k1` ECDSA curve recommendation. Any randomly chosen 256-bit number from `0x1` to `0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140` is a valid private key.

Private keys are usually encoded using **Wallet Import Format** (**WIF**) in order to make them easier to copy and use. It is a way to represent the full size private key in a different format. WIF can be converted into a private key and vice versa. The steps are described here.

The following is an example of a private key:

```
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA5714195201
```

When it is converted into WIF format it looks like this:

```
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP
```

Interested readers can do some experimentation using the tool available at the following website:
`http://gobittest.appspot.com/PrivateKey`

Also, **mini private key format** is sometimes used to create the private key with a maximum of up to 30 characters in order to allow storage where physical space is limited, for example, etching on physical coins or encoding in damage-resistant QR codes. The QR code becomes more damage resistant because more dots can be used for error correction and less for encoding the private key. The private key encoded using mini private key format is also sometimes called **minikey**. The first character of mini private key is always uppercase letter `S`. A mini private key can be converted into a normal size private key but an existing normal size private key cannot be converted into a mini private key. This format was used in Casascius physical bitcoins.

Interested readers can find more information here
`https://en.bitcoin.it/wiki/Casascius_physical_bitcoins`.

A Casascius physical bitcoin's security hologram paper with minikey and QR code

The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

Public keys in Bitcoin

Public keys exist on the blockchain and all network participants can see it. Public keys are derived from private keys due to their special mathematical relationship with the private keys. Once a transaction signed with the private key is broadcasted on the Bitcoin network, public keys are used by the nodes to verify that the transaction has indeed been signed with the corresponding private key. This process of verification proves the ownership of the bitcoin.

Bitcoin uses ECC based on the `secp256k1` standard. More specifically it makes use of ECDSA to ensure that funds remain secure and can only be spent by the legitimate owner. If you need to refresh the relevant cryptography concepts, you can refer to `Chapter 4`, *Public Key Cryptography* where ECC was explained. A public key is 256-bits in length. Public keys can be represented in an uncompressed or compressed format. Public keys are fundamentally x and y coordinates on an elliptic curve. In an uncompressed format public keys are presented with a prefix of `0x4` in a hexadecimal format. The x and y coordinates are both 32-bit in length. In total, the compressed public key is 33-bytes long as compared to 65-bytes in the uncompressed format. The compressed version of public keys includes only the x part, since the y part can be derived from it.

The reason why the compressed version of public keys works is that if the ECC graph is visualized, it reveals that the *y* coordinate can be either below the *x* axis or above the *x* axis and as the curve is symmetric, only the location in the prime field is required to be stored. If *y* is even then it is above the *x* axis and if it is odd then it is below the *x* axis. This means that instead of storing both *x* and *y* as the public key only *x* can be stored with the information that if *y* is even or odd.

Initially, Bitcoin client used uncompressed keys, but starting from Bitcoin core client 0.6, compressed keys are used as standard. This resulted in almost 50% reduction of space used to store public keys in the blockchain.

Keys are identified by various prefixes, described as follows:

- Uncompressed public keys use `0x04` as the prefix
- Compressed public key starts with `0x03` if the *y* 32-bit part of the public key is odd
- Compressed public key starts with `0x02` if the *y* 32-bit part of the public key is even

Addresses in Bitcoin

A bitcoin address is created by taking the corresponding public key of a private key and hashing it twice, first with the SHA-256 algorithm and then with RIPEMD-160. The resultant 160-bit hash is then prefixed with a version number and finally encoded with a Base58Check encoding scheme. The bitcoin addresses are 26-35 characters long and begin with digit `1` or `3`.

A typical bitcoin address looks like a string shown here:

```
1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt
```

This is also commonly encoded in a QR code for easy distribution. The QR code of the preceding bitcoin address is shown in the following screenshot:

QR code of a bitcoin address 1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

Currently, there are two types of addresses, the commonly used P2PKH and another P2SH type, starting with number `1` and `3`, respectively. In the early days, Bitcoin used direct Pay to Pubkey, which is now superseded by P2PKH. These types will be explained later in the chapter. However, direct Pay to Pubkey is still used in Bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise. Avoiding address reuse circumvents anonymity issues to an extent, Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks and selfish mining which require different approaches to resolve.

Transaction malleability has been resolved with so-called *Segregated Witness* soft fork upgrade of the Bitcoin protocol. This concept will be explained later in the chapter.

From bitaddress.org, private key and bitcoin address in a paper wallet

Base58Check encoding

Bitcoin addresses are encoded using the Base58Check encoding. This encoding is used to limit the confusion between various characters, such as 0OIl as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utilizing a set of 58 alphanumeric symbols. More explanation and logic can be found in the `base58.h` source file (`https://github.com/bitcoin/bitcoin/blob/master/src/base58.h`) in the bitcoin source code:

```
/**
* Why base-58 instead of standard base-64 encoding?
* - Don't want 0OIl characters that look the same in some fonts and
* could be used to create visually identical looking data.
* - A string with non-alphanumeric characters is not as easily accepted as
input.
* - E-mail usually won't line-break if there's no punctuation to break at.
* - Double-clicking selects the whole string as one word if it's all
alphanumeric.
*/
```

Vanity addresses

As bitcoin addresses are based on base-58 encoding, it is possible to generate addresses that contain human-readable messages. An example is shown as follows:

Vanity public address encoded in QR

Vanity addresses are generated using a purely brute-force method. An example of a paper wallet with vanity address is shown in the following screenshot:

Vanity address generated from https://bitcoinvanitygen.com/

In the preceding screenshot, on the right-hand bottom corner the public vanity address with QR code is displayed. The paper wallets can be stored physically as an alternative to electronic storage of private keys.

Multisignature addresses

As the name implies, these addresses require multiple private keys. In practical terms, it means that in order to release the coins a certain set of signatures is required. This is also known as **M-of-N MultiSig**. Here *M* represents threshold or the minimum number of signatures required from *N* number of keys to release the bitcoins.

Transactions

Transactions are at the core of the bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a bitcoin address, or it can be quite complex depending on the requirements. Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction and outputs as coins being created. If a transaction is minting new coins, then there is no input and therefore no signature is needed. If a transaction is to send coins to some other user (a bitcoin address), then it needs to be signed by the sender with their private key and a reference is also required to the previous transaction in order to show the origin of the coins. Coins are, in fact, unspent transaction outputs represented in Satoshis.

Transactions are not encrypted and are publicly visible in the blockchain. Blocks are made up of transactions and these can be viewed using any online blockchain explorer.

The transaction life cycle

The following steps describe the transaction life cycle:

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcasted to the Bitcoin network using a flooding algorithm.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transaction are placed in the block they are placed in a special memory buffer called **transaction pool**. The purpose of the transaction pool is explained in the next section.
5. Mining starts, which is a process by which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources. This concept is explained in more detail later in this chapter.
6. Once a miner solves the PoW problem it broadcasts the newly mined block to the network. PoW is explained in detail later in this chapter.
7. The nodes verify the block and propagate the block further, and confirmations start to generate.

8. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed. However, three to six is just a recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after three confirmations.

Transaction fee

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs and the sum of the outputs.

A simple formula can be used:

$$fee = sum(inputs) - sum(outputs)$$

The fees are used as an incentive for miners to encourage them to include a user transaction in the block the miners are creating. All transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners.

There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes. Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time. This is however no longer practical due to the high volume of transactions and competing investors on the Bitcoin network, therefore it is advisable to provide a fee always. The time for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. Transaction time is dependent on transaction fees and network activity. If the network is very busy then naturally transactions will take longer to process and if you pay a higher fee then your transaction is more likely to be picked by miners first due to additional incentive of the higher fee.

Transaction pools

Also known as memory pools, these pools are basically created in local memory (computer RAM) by nodes in order to maintain a temporary list of transactions that are not yet confirmed in a block. Transactions are included in a block after passing verification and based on their priority.

The transaction data structure

A transaction at a high level contains metadata, inputs, and outputs. Transactions are combined to create a block.

The transaction data structure is shown in the following table:

Field	**Size**	**Description**
Version number	4 bytes	Used to specify rules to be used by the miners and nodes for transaction processing.
Input counter	1-9 bytes	The number (positive integer) of inputs included in the transaction.
List of inputs	Variable	Each input is composed of several fields, including `Previous Tx hash`, `Previous Txout-index`, `Txin-script length`, `Txin-script`, and optional sequence number. The first transaction in a block is also called a coinbase transaction. It specifies one or more transaction inputs.
Output counter	1-9 bytes	A positive integer representing the number of outputs.
List of outputs	Variable	Outputs included in the transaction.
Lock time	4 bytes	This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or block height.

A sample transaction is shown as follows. This is the decoded transaction from the first example of a payment transaction provided at the start of this chapter.

```
{
   "lock_time":0,
   "size":226,
   "inputs":[
      {
```

```
         "prev_out":{
            "index":139,
"hash":"40120e43f00ff96e098a9173f14f1371655b3478bc0a558d6dc17a4ab176387d"
         },
"script":"483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7af
bf92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71
758d012102c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12"
      }
   ],
   "version":1,
   "vin_sz":1,
"hash":"d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d",
   "vout_sz":2,
   "out":[
      {
         "script_string":"OP_DUP OP_HASH160
c568ffeb46c6a9362e44a5a49deaa6eab05a619a OP_EQUALVERIFY OP_CHECKSIG",
         "address":"1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3",
         "value":33324,
         "script":"76a914c568ffeb46c6a9362e44a5a49deaa6eab05a619a88ac"
      },
      {
         "script_string":"OP_DUP OP_HASH160
9386c8c880488e80a6ce8f186f788f3585f74aee OP_EQUALVERIFY OP_CHECKSIG",
         "address":"1ET3oBGf8JpunjytE7owyVtmBjmvcDycQe",
         "value":93376,
         "script":"76a9149386c8c880488e80a6ce8f186f788f3585f74aee88ac"
      }
   ]
}
```

As shown in the preceding code, there are a number of structures that make up the transaction. All these elements are described in the following subsections.

Metadata

This part of the transaction contains some values such as the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a `lock_time` field. Every transaction has a prefix specifying the version number. These fields are shown in the preceding example: `lock_time`, `size`, and `version`.

Inputs

Generally, each input spends a previous output. Each output is considered as **Unspent Transaction Output** (**UTXO**) until an input consumes it. UTXO is an unspent transaction output that can be spent as an input to a new transaction.

Transaction input data structure is shown in the following table:

Field	**Size**	**Description**
Transaction hash	32 bytes	This is the hash of the previous transaction with UTXO.
Output index	4 bytes	This is the previous transactions output index, that is, UTXO to be spent.
Script length	1-9 bytes	This is the size of the unlocking script.
Unlocking script	Variable	Input script (`ScriptSig`) which satisfies the requirements of the locking script.
Sequence number	4 bytes	Usually disabled or contains lock time. Disabled is represented by `'0xFFFFFFFF'`.

In the preceding example the inputs are defined under `"inputs" : [` section.

Outputs

Outputs have three fields, and they contain instructions for sending bitcoins. The first field contains the amount of Satoshis whereas the second field contains the size of the locking script. Finally, the third field contains a locking script that holds the conditions that need to be met in order for the output to be spent. More information on transaction spending using locking and unlocking scripts and producing outputs is discussed later in this section.

Transaction output data structure is shown here:

Field	**Size**	**Description**
Value	8 bytes	Total number in positive integers of Satoshis to be transferred
Script size	1-9 bytes	Size of the locking script
Locking script	Variable	Output script (`ScriptPubKey`)

In the preceding example two outputs are shown under `"OUT":[` section.

Verification

Verification is performed using Bitcoin's scripting language which is described in the next section in detail.

The script language

Bitcoin uses a simple stack-based language called **script** to describe how bitcoins can be spent and transferred. It is not Turing complete and has no loops to avoid any undesirable effects of long-running/hung scripts on the Bitcoin network. This scripting language is based on a Forth programming language like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from the left to the right using a **Last In, First Out** (**LIFO**) stack.

Scripts use various opcodes or instructions to define their operation. Opcodes are also known as words, commands, or functions. Earlier versions of the Bitcoin node had a few opcodes that are no longer used due to bugs discovered in their design.

The various categories of the scripting opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography, and lock time.

A transaction script is evaluated by combining `ScriptSig` and `ScriptPubKey`. `ScriptSig` is the unlocking script, whereas `ScriptPubKey` is the locking script. This is how a transaction to be spent is evaluated:

1. First, it is unlocked and then it is spent
2. `ScriptSig` is provided by the user who wishes to unlock the transaction
3. `ScriptPubkey` is part of the transaction output and specifies the conditions that need to be fulfilled in order to spend the output
4. In other words, outputs are locked by `ScriptPubKey` that contains the conditions, when met will unlock the output, and coins can then be redeemed

Commonly used opcodes

All opcodes are declared in the `script.h` file in the Bitcoin reference client source code.

This can be accessed from the link at `https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h` under the following comment:
`/** Script opcodes */`

A description of the most commonly used opcodes is listed here. This table is taken from the Bitcoin developer's guide:

Opcode	Description
`OP_CHECKSIG`	This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then `TRUE` is pushed onto the stack; otherwise, `FALSE` is pushed.
`OP_EQUAL`	This returns `1` if the inputs are exactly equal; otherwise, `0` is returned.
`OP_DUP`	This duplicates the top item in the stack.
`OP_HASH160`	The input is hashed twice, first with SHA-256 and then with RIPEMD-160.
`OP_VERIFY`	This marks the transaction as invalid if the top stack value is not true.
`OP_EQUALVERIFY`	This is the same as `OP_EQUAL`, but it runs `OP_VERIFY` afterwards.
`OP_CHECKMULTISIG`	This takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of `1` is returned as a result; otherwise, `0` is returned.

Types of transactions

There are various scripts available in Bitcoin to handle the value transfer from the source to the destination. These scripts range from very simple to quite complex depending upon the requirements of the transaction. Standard transaction types are discussed here. Standard transactions are evaluated using `IsStandard()` and `IsStandardTx()` tests and only standard transactions that pass the test are generally allowed to be mined or broadcasted on the Bitcoin network. However, nonstandard transactions are valid and allowed on the network.

The following are the standard transaction types:

- **Pay to Public Key Hash** (**P2PKH**): P2PKH is the most commonly used transaction type and is used to send transactions to the bitcoin addresses. The format of the transaction is shown as follows:

```
      ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG
      ScriptSig: <sig> <pubKey>
```

  The `ScriptPubKey` and `ScriptSig` parameters are concatenated together and executed. An example will follow shortly in this section, where this is explained in more detail.

- **Pay to Script Hash** (**P2SH**): P2SH is used in order to send transactions to a script hash (that is, the addresses starting with `3`) and was standardized in BIP16. In addition to passing the script, the redeem script is also evaluated and must be valid. The template is shown as follows:

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL
ScriptSig: [<sig>...<sign>] <redeemScript>
```

- **MultiSig** (**Pay to MultiSig**): M-of-N MultiSig transaction script is a complex type of script where it is possible to construct a script that required multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig > . . . <sign>]
```

  Raw multisig is obsolete, and multisig is usually part of the P2SH redeem script, mentioned in the previous bullet point.

- **Pay to Pubkey**: This script is a very simple script that is commonly used in coinbase transactions. It is now obsolete and was used in an old version of bitcoin. The public key is stored within the script in this case, and the unlocking script is required to sign the transaction with the private key.

  The template is shown as follows:

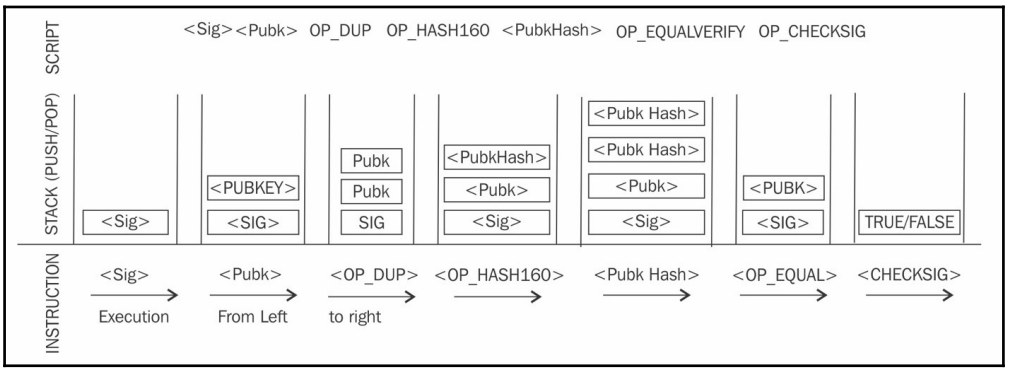
```
<PubKey> OP_CHECKSIG
```

- **Null data/OP_RETURN**: This script is used to store arbitrary data on the blockchain for a fee. The limit of the message is 40 bytes. The output of this script is unredeemable because `OP_RETURN` will fail the validation in any case. `ScriptSig` is not required in this case.

The template is very simple and is shown as follows:

```
OP_RETURN <data>
```

A P2PKH script execution is shown in the following diagram:



P2PKH script execution

All transactions are eventually encoded into the hexadecimal format before transmitting over the Bitcoin network. A sample transaction is shown here in hexadecimal format that is retrieved using `bitcoin-cli` on the Bitcoin node running on mainnet:

```
$ bitcoin-cli getrawtransaction
"d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d"
{
  "result":
"01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef90ff0430e1240
8b0000006b483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7af
bf92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71
758d012102c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12f
ffffffff022c82000000000001976a914c568ffeb46c6a9362e44a5a49deaa6eab05a619a88
acc06c0100000000001976a9149386c8c880488e80a6ce8f186f788f3585f74aee88ac00000
000",
  "error": null,
  "id": null
}
```

Note that this is the same transaction that was presented as an example at the start of this chapter.

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called `coinbase`, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data that can be used to store arbitrary data. In the genesis block, this transaction included the most famous comment taken from *The Times* newspaper:

> *"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks."*

This message is a proof that the genesis block was not mined earlier than January 3, 2009. This is because first Bitcoin block (genesis block) was created on January 3, 2009 and this news excerpt was taken from that day's newspaper.

A coinbase transaction input has the same number of fields as usual transaction input, but the structure contains coinbase data size and coinbase data fields instead of unlocking script size and unlocking script fields. Also, it does not have a reference pointer to the previous transaction. This structure is shown in the following table:

Field	**Size**	**Description**
Transaction hash	32 bytes	Set to all zeroes as no hash reference is used
Output index	4 bytes	Set to `0xFFFFFFFF`
Coinbase data length	1-9 bytes	2 bytes-100 bytes
Data	Variable	Any data
Sequence number	4 bytes	Set to `0xFFFFFFFF`

Contracts

As defined in the Bitcoin core developer guide, contracts are basically transactions that use the Bitcoin system to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to design complex contracts that can be used in many real-world scenarios. Contracts allow the development of a completely decentralized, independent, and reduced risk platform.

Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language. The current implementation of a script is very limited, but various types of contracts are still possible to develop. For example, the release of funds only if multiple parties sign the transaction or perhaps the release of funds only after a certain time has elapsed. Both of these scenarios can be realized using multisig and transaction lock time options.

Transaction verification

This verification process is performed by Bitcoin nodes. The following is described in the Bitcoin developer guide:

1. Check the syntax and ensure that the syntax and data structure of the transaction conforms to the rules provided by the protocol.
2. Verify that no transaction inputs and outputs are empty.
3. Check whether the size in bytes is less than the maximum block size.
4. The output value must be in the allowed money range (0 to 21 million BTC).
5. All inputs must have a specified previous output, except for coinbase transactions, which should not be relayed.
6. Verify that `nLockTime` must not exceed 31-bits. (`nLockTime` specifies the time before which transaction will not be included in the block.)
7. For a transaction to be valid, it should not be less than 100 bytes.
8. The number of signature operations in a standard transaction should be less than or not more than two.
9. Reject nonstandard transactions; for example, `ScriptSig` is allowed to only push numbers on the stack. `ScriptPubkey` not passing the `isStandard()` checks. The `isStandard()` checks specify that only standard transactions are allowed.
10. A transaction is rejected if there is already a matching transaction in the pool or in a block in the main branch.
11. The transaction will be rejected if the referenced output for each input exists in any other transaction in the pool.
12. For each input, there must exist a referenced output unspent transaction.
13. For each input, if the referenced output transaction is the coinbase, it must have at least 100 confirmations; otherwise, the transaction will be rejected.
14. For each input, if the referenced output does not exist or has been spent already, the transaction will be rejected.

15. Using the referenced output transactions to get input values, verify that each input value, as well as the sum, is in the allowed range of 0-21 million BTC. Reject the transaction if the sum of input values is less than the sum of output values.
16. Reject the transaction if the transaction fee would be too low to get into an empty block.
17. Each input unlocking script must have corresponding valid output scripts.

Transaction malleability

Transaction malleability in Bitcoin was introduced due to a bug in the bitcoin implementation. Due to this bug, it became possible for an adversary to change the transaction ID of a transaction, thus resulting in a scenario where it would appear that a certain transaction has not been executed. This can allow scenarios where double deposits or withdrawals can occur. In other words, this bug allows the changing of the unique ID of a Bitcoin transaction before it is confirmed. If the ID is changed before confirmation, it would seem that the transaction did not occur at all which can then allow these attacks.

Blockchain

Blockchain is a public ledger of a timestamped, ordered, and immutable list of all transactions on the Bitcoin network. Each block is identified by a hash in the chain and is linked to its previous block by referencing the previous block's hash.

In the following table structure of a block is presented, followed by a detailed diagram that provides a detailed view of the blockchain structure.

The structure of a block

The following table shows the structure of a block:

Field	Size	Description
Block size	4 bytes	This is the size of the block.
Block header	80 bytes	This includes fields from the block header described in the next section.

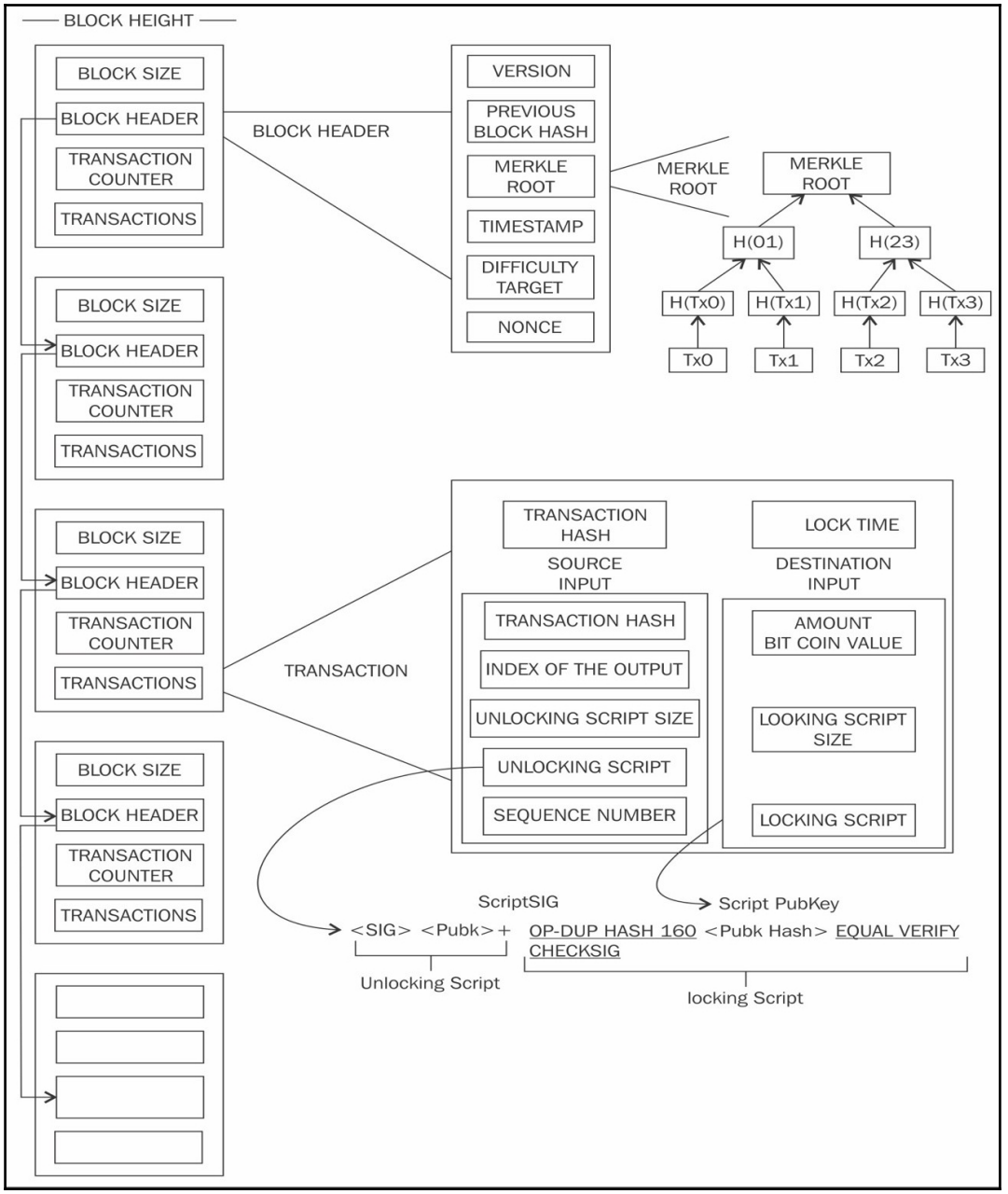
Transaction counter	Variable	This field contains the total number of transactions in the block, including the coinbase transaction. Size ranges from 1-9 bytes
Transactions	Variable	All transactions in the block.

The structure of a block header

The following table depicts the structure of a block header:

Field	**Size**	**Description**
Version	4 bytes	The block version number that dictates the block validation rules to follow.
Previous block's header hash	32 bytes	This is a double SHA-256 hash of the previous block's header.
Merkle root hash	32 bytes	This is a double SHA-256 hash of the Merkle tree of all transactions included in the block.
Timestamp	4 bytes	This field contains the approximate creation time of the block in the Unix epoch time format. More precisely, this is the time when the miner has started hashing the header. (The time from the miner's point of view.)
Difficulty target	4 bytes	This is the current difficulty target of the network/block.
Nonce	4 bytes	This is an arbitrary number that miners change repeatedly to produce a hash that is lower than the difficulty target.

As shown in the following diagram, blockchain is a chain of blocks where each block is linked to its previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block.



A visualization of the blockchain, block, block header, transactions and scripts

The preceding diagram shows a high-level overview of the Bitcoin blockchain. On the left-hand side blocks are shown starting from top to bottom. Each block contains transactions and block headers which are further magnified on the right-hand side. On the top, first, block header is expanded to show various elements within the block header. Then on the right-hand side the Merkle root element of the block header is shown in magnified view which shows that how Merkle root is calculated. We have discussed Merkle trees in detail previously, you can refer to `Chapter 3`, *Symmetric Cryptography* if you need to revise the concept. Further down transactions are also magnified to show the structure of a transaction and the elements that it contains. Also, note that transactions are then further elaborated by showing that what locking and unlocking scripts look like. This diagram shows a lot of components, we will discuss all these in this chapter.

The genesis block

This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the bitcoin core software. It is in the `chainparams.cpp` file (`https://github.com/bitcoin/bitcoin/blob/master/src/chainparams.cpp`):

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t
nBits, int32_t nVersion, const CAmount& genesisReward)
{
   const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of
second bailout for banks";
   const CScript genesisOutputScript = CScript() <<
ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb
649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f") <<
OP_CHECKSIG;
   return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime,
nNonce,
   nBits, nVersion, genesisReward);
}
```

Bitcoin provides protection against double spending by enforcing strict rules on transaction verification and via mining. Transactions and blocks are added in the blockchain only after strict rule checking explained earlier in the *Transaction verification* section and successful PoW solution. Block height is the number of blocks before a particular block in the blockchain. The current height (as of March 6, 2018) of the blockchain is 512,328 blocks. PoW is used to secure the blockchain. Each block contains one or more transactions, out of which the first transaction is a coinbase transaction. There is a special condition for coinbase transactions that prevent them from being spent until at least 100 blocks in order to avoid a situation where the block may be declared stale later on.

Stale blocks are created when a block is solved and every other miner who is still working to find a solution to the hash puzzle is working on that block. Mining and hash puzzles will be discussed later in the chapter in detail. As the block is no longer required to be worked on, this is considered a stale block.

Orphan blocks are also called detached blocks and were accepted at one point in time by the network as valid blocks but were rejected when a proven longer chain was created that did not include this initially accepted block. They are not part of the main chain and can occur at times when two miners manage to produce the blocks at the same time.

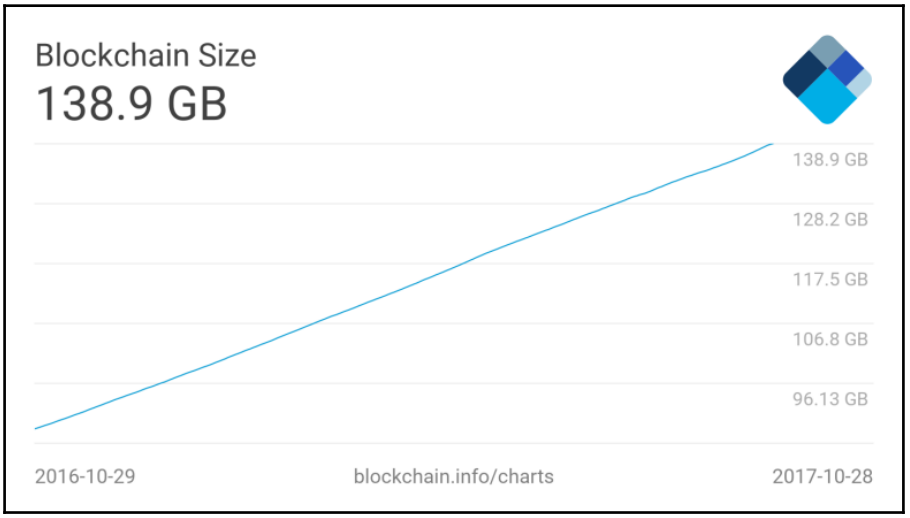
The latest block version is version 4, which was proposed with BIP65 and has been used since bitcoin core client 0.11.2 since the implementation of BIP9 bits in the `nVersion` field are being used to indicate soft fork changes.

Because of the distributed nature of bitcoin, network forks can occur naturally. In cases where two nodes simultaneously announce a valid block can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the Bitcoin network only by accepting the longest chain. In this case, the smaller chain will be considered orphaned. If an adversary manages to gain 51% control of the network hash rate (computational power), then they can impose their own version of transaction history.

Forks in blockchain can also occur with the introduction of changes in the Bitcoin protocol. In case of a *soft fork*, a client which chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, previous and new blocks are both acceptable, thus making soft fork backwards compatible.

In case of a soft fork, only miners are required to upgrade to the new client software in order to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated already. A hard fork, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure change or major protocol changes results in a hard fork. The current size of the bitcoin blockchain as of October 29, 2017, stands at approximately 139 GB.

The following diagram shows the size increase of blockchain as a function of time:



Current size of blockchain as of 29/10/2017

New blocks are added to the blockchain approximately every 10 minutes and network difficulty is adjusted dynamically every 2016 blocks in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following equation:

$$Target = Previous\ target * Time/2016 * 10\ minutes$$

Difficulty and target are interchangeable and represent the same thing. Previous target represents the old target value, and time is the time spent to generate previous 2016 blocks. Network difficulty basically means how hard it is for miners to find a new block, that is, how difficult the hashing puzzle is now.

In the next section, mining is discussed, which will explain how the hashing puzzle is solved.

Mining

Mining is a process by which new blocks are added to the blockchain. Blocks contain transactions that are validated via the mining process by mining nodes on the Bitcoin network. Blocks, once mined and verified are added to the blockchain which keeps the blockchain growing. This process is resource-intensive due to the requirements of PoW where miners compete in order to find a number which is less than the difficulty target of the network. This difficulty in finding the correct value (also called sometimes the mathematical puzzle) is there to ensure that the required resources have been spent by miners before a new proposed block can be accepted. New coins are minted by the miners by solving the PoW problem, also known as partial hash inversion problem. This process consumes a high amount of resources including computing power and electricity. This process also secures the system against frauds and double spending attacks while adding more virtual currency to the Bitcoin ecosystem.

Roughly one new block is created (mined) every 10 minutes to control the frequency of generation of bitcoins. This frequency needs to be maintained by the Bitcoin network and is encoded in the bitcoin core clients in order to control the *money supply*. Miners are rewarded with new coins if and when they discover new blocks by solving PoW. Miners are paid transaction fees in return for including transactions in their proposed blocks. New blocks are created at an approximate fixed rate of every 10 minutes. The rate of creation of new bitcoins decreases by 50%, every 210,000 blocks, roughly every 4 years. When bitcoin was initially introduced, the block reward was 50 bitcoins; then in 2012, this was reduced to 25 bitcoins. In July 2016, this was further reduced to 12.5 coins (12 coins) and the next reduction is estimated to be on July 4, 2020. This will reduce the coin reward further down to approximately six coins.

Approximately 144 blocks, that is, 1,728 bitcoins are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at 144 per day. Bitcoin supply is also limited and in 2140, almost 21 million bitcoins will be finally created and no new bitcoins can be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

Tasks of the miners

Once a node connects to the bitcoin network, there are several tasks that a bitcoin miner performs:

1. **Synching up with the network**: Once a new node joins the bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the bitcoin miner; however, this not necessarily a task only for a miner.
2. **Transaction validation**: Transactions broadcasted on the network are validated by full nodes by verifying and validating signatures and outputs.
3. **Block validation**: Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with verification of the nonce value.
4. **Create a new block**: Miners propose a new block by combining transactions broadcasted on the network after validating them.
5. **Perform Proof of Work**: This task is the core of the mining process and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.
6. **Fetch reward**: Once a node solves the hash puzzle (PoW), it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners on the network due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with 12.5 bitcoins and any associated transaction fees.

Mining rewards

When Bitcoin started in 2009 the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012 it halved down to 25 bitcoins. Currently, it is 12.5 BTC per block since July 2016. Next halving is on Friday, 12 June 2020 after which the block reward will be reduced down to 6.25 BTC per block. This mechanism is hardcoded in Bitcoin to regulate, control inflation and limit the supply of bitcoins.

Proof of Work (PoW)

This is a proof that enough computational resources have been spent in order to build a valid block. PoW is based on the idea that a random node is selected every time to create a new block. In this model, nodes compete with each other in order to be selected in proportion to their computing capacity. The following equation sums up the PoW requirement in bitcoin:

$$H (N \mid\mid P\_hash \mid\mid Tx \mid\mid Tx \mid\mid \ldots Tx) < Target$$

Where *N* is a nonce, *P_hash* is a hash of the previous block, *Tx* represents transactions in the block, and *Target* is the target network difficulty value. This means that the hash of the previously mentioned concatenated fields should be less than the target hash value.

The only way to find this nonce is the brute force method. Once a certain pattern of a certain number of zeroes is met by a miner, the block is immediately broadcasted and accepted by other miners.

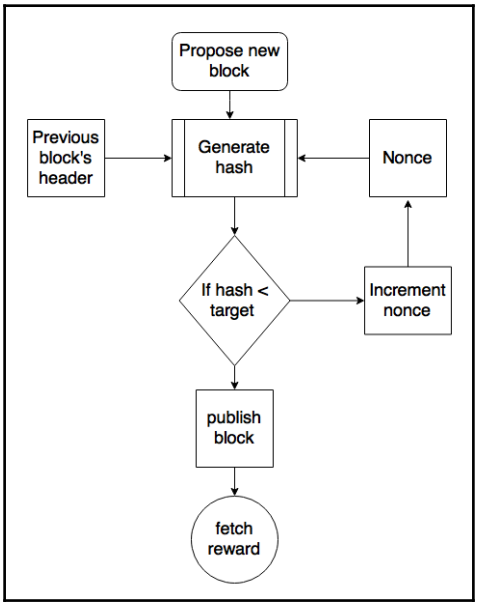
The mining algorithm

The mining algorithm consists of the following steps.

1. The previous block's header is retrieved from the bitcoin network.
2. Assemble a set of transactions broadcasted on the network into a block to be proposed.
3. Compute the double hash of the previous block's header combined with a nonce and the newly proposed block using the SHA-256 algorithm.
4. Check if the resultant hash is lower than the current difficulty level (target) then PoW is solved. As a result of successful PoW the discovered block is broadcasted to the network and miners fetch the reward.
5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce.

   As the hash rate of the bitcoin network increased, the total amount of 32-bit nonce was exhausted too quickly. In order to address this issue, the extra nonce solution was implemented, whereby the coinbase transaction is used as a source of extra nonce to provide a larger range of nonce to be searched by the miners.

This process can be visualized by using the following flowchart:



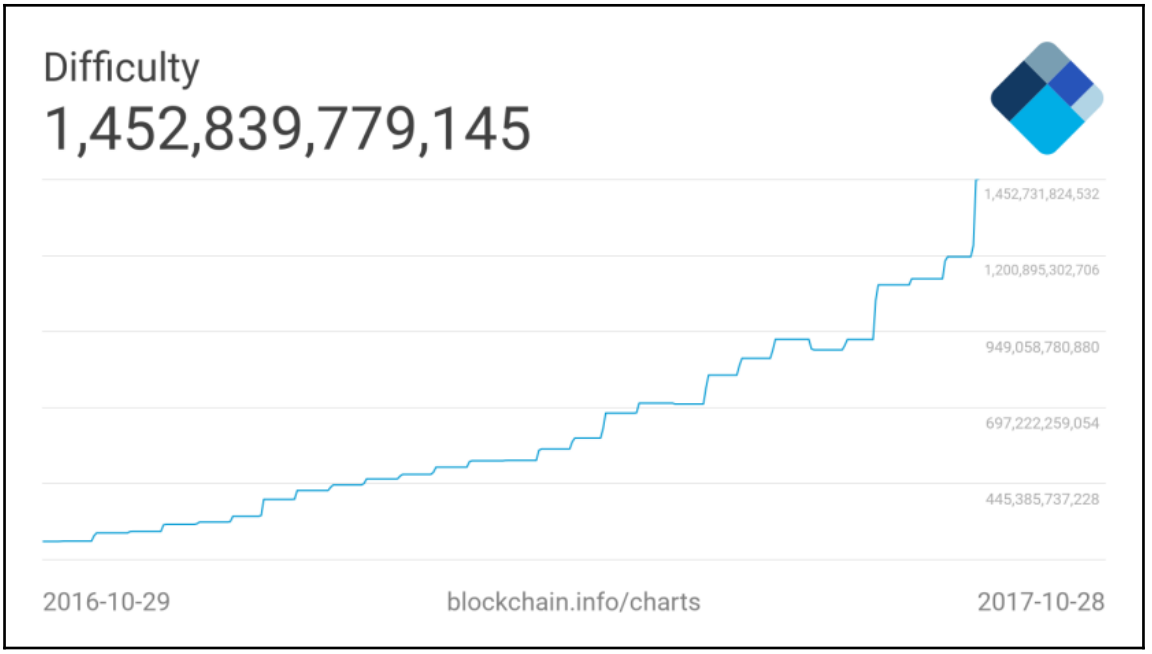
Mining process

Mining difficulty increased over time and bitcoins that could be mined by single CPU laptop computers now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried using the Bitcoin command-line interface using the following command:

```
$ bitcoin-cli getdifficulty
1452839779145
```

This number represents the difficulty level of the Bitcoin network. Recall from previous sections that miners compete to find a solution to a problem. This number, in fact shows, that how difficult it is to find a hash which is lower than the network difficulty target. All successfully mined blocks must contain a hash that is less than this target number. This number is updated every 2 weeks or 2016 blocks to ensure that on average 10-minute block generation time is maintained.

Bitcoin network difficulty has increased exponentially, the following graph shows this difficulty level over a period of one year:

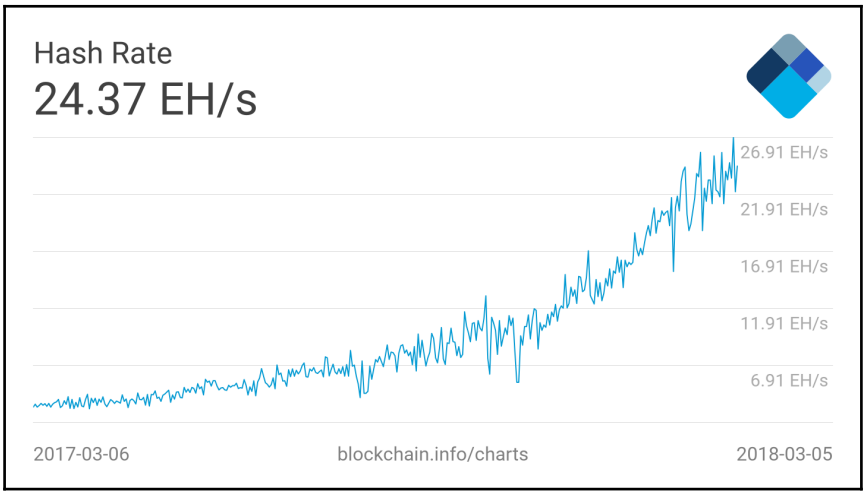


Mining difficulty over the last year

The preceding graph shows the difficulty of the Bitcoin network over a period of last year and it has increased quite significantly. The reason why mining difficulty increases is because in Bitcoin, the block generation time has to be always around 10 minutes. This means that if blocks are being mined too quickly by fast hardware then the difficulty increases so that the block generation time can remain at roughly 10 minutes per block. This is also true in reverse if blocks are not mined every 10 minutes than the difficulty is decreased. Difficulty, is calculated every 2016 blocks (in two weeks) and adjusted accordingly. If the previous set of 2016 blocks were mined in less than a period of two weeks then difficulty will be increased. Similarly, if 2016 blocks were found in more than two weeks (If blocks are mined every 10 minutes then 2016 blocks take 2 weeks to be mined) then the difficulty is decreased.

The hash rate

The hashing rate basically represents the rate of calculating hashes per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block. In early days of bitcoin, it used to be quite small as CPUs were used. However, with dedicated mining pools and ASICs now, this has gone up exponentially in the last few years. This has resulted in increased difficulty of the Bitcoin network. The following hash rate graph shows the hash rate increase over time and is currently measured in Exa hashes. This means that in 1 second, the Bitcoin network miners are computing more than 24,000,000,000,000,000,000 hashes per second.



Hashing rate (measured in Exa-hashes) as of March 2018, shown over a period of 1 year

Mining systems

Over time, bitcoin miners have used various methods to mine bitcoins. As the core principle behind mining is based on the double SHA-256 algorithm, overtime experts have developed sophisticated systems to calculate the hash faster and faster. The following is a review of the different types of mining methods used in bitcoin and how they evolved with time.

CPU

CPU mining was the first type of mining available in the original bitcoin client. Users could even use laptop or desktop computers to mine bitcoins. CPU mining is no longer profitable and now more advanced mining methods such as ASIC-based mining is used. CPU mining only lasted for around just over a year since the introduction of Bitcoin and soon other methods were explored and tried by the miners.

GPU

Due to the increased difficulty of the bitcoin network and the general tendency of finding faster methods to mine, miners started to use GPUs or graphics cards available in PCs to perform mining. GPUs support faster and parallelized calculations that are usually programmed using the OpenCL language. This turned out to be a faster option as compared to CPUs. Users also used techniques such as overclocking to gain maximum benefit of the GPU power. Also, the possibility of using multiple graphics cards increased the popularity of graphics cards' usage for bitcoin mining. GPU mining, however, has some limitations, such as overheating and the requirement for specialized motherboards and extra hardware to house multiple graphics cards. From another angle, graphics cards have become quite expensive due to increased demand and this has impacted gamers and graphic software users.

FPGA

Even GPU mining did not last long, and soon miners found another way to perform mining using FPGAs. **Field Programmable Gate Array** (**FPGA**) is basically an integrated circuit that can be programmed to perform specific operations. FPGAs are usually programmed in **Hardware Description Languages** (**HDLs**), such as Verilog and VHDL. Double SHA-256 quickly became an attractive programming task for FPGA programmers and several open source projects started too. FPGA offered much better performance as compared to GPUs; however, issues such as accessibility, programming difficulty, and the requirement for specialized knowledge to program and configure FPGAs resulted in a short life of the FPGA era for bitcoin mining.

The arrival of ASICs resulted in quickly phased out FPGA- based systems for mining. Mining hardware such as X6500 miner, Ztex, and Icarus were developed during the time when FPGA mining was profitable. Various FPGA manufacturers, such as Xilinx and Altera, produce FPGA hardware and development boards that can be used to program mining algorithms. It should be noted that GPU mining is still profitable for some other cryptocurrencies to some extent such as Zcoin (`https://zcoin.io/guide-on-how-to-mine-zcoin-xzc/`), but not Bitcoin, because network difficulty of Bitcoin is so high that only ASICs (specialized hardware) running in parallel can produce some reasonable profit.

ASICs

Application Specific Integrated Circuit (**ASIC**) was designed to perform the SHA-256 operation. These special chips were sold by various manufacturers and offered a very high hashing rate. This worked for some time, but due to the quickly increasing mining difficulty level, single-unit ASICs are no longer profitable.

Currently, mining is out of the reach of individuals as vast amounts of energy and money is needed to be spent in order to build a profitable mining platform. Now professional mining centers using thousands of ASIC units in parallel are offering mining contracts to users to perform mining on their behalf. There is no technical limitation, a single user can run thousands of ASICs in parallel but it will require dedicated data centers and hardware, therefore, cost for a single individual can become prohibitive. The following are the four types of mining hardware:

CPU

GPU

FPGA

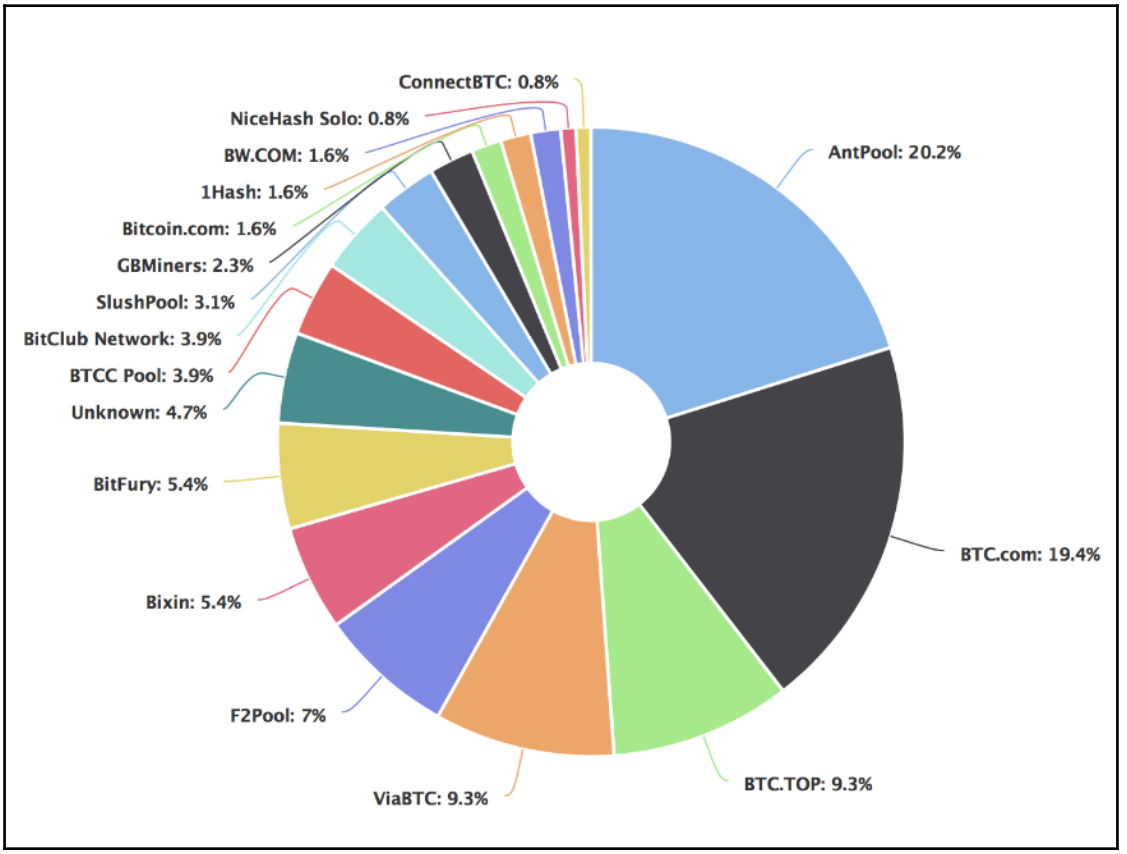
ASIC

Mining pools

A mining pool forms when group of miners work together to mine a block. The pool manager receives the coinbase transaction if the block is successfully mined, which is then responsible for distributing the reward to the group of miners who invested resources to mine the block. This is profitable as compared to solo mining, where only one sole miner is trying to solve the partial hash inversion function (hash puzzle) because, in mining pools, the reward is paid to each member of the pool regardless of whether they (more specifically, their individual node) solved the puzzle or not.

There are various models that a mining pool manager can use to pay to the miners, such as the **Pay Per Share** (**PPS**) model and the proportional model. In the PPS model, the mining pool manager pays a flat fee to all miners who participated in the mining exercise, whereas in the proportional model, the share is calculated based on the amount of computing resources spent to solve the hash puzzle.

Many commercial pools now exist and provide mining service contracts via the cloud and easy-to-use web interfaces. The most commonly used ones are AntPool (`https://www.antpool.com`), BTC (`https://btc.com`), and BTC TOP (`http://www.btc.top`). A comparison of hashing power for all major mining pools is shown in the following diagram:



Mining pools and their hashing power (hash rate) as of 28/10/2017

Source: https://blockchain.info/pools

Mining centralization can occur if a pool manages to control more than 51% of the network by generating more than 51% hash rate of the Bitcoin network.

As discussed earlier in the introduction section, a 51% attack can result in successful double-spending attacks, and it can impact consensus and in fact impose another version of transaction history on the Bitcoin network.

This event has happened once in the Bitcoin history when GHash.IO, a large mining pool, managed to acquire more than 51% of the network capacity. Theoretical solutions, such as two-phase PoW (`http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/`), have been proposed in academia to disincentivize large mining pools. This scheme introduces a second cryptographic puzzle that results in mining pools to either reveal their private keys or provide a considerable portion of the hash rate of their mining pool, thus reducing the overall hash rate of the pool.

Various types of hardware are commercially available for mining purposes. Currently, the most profitable one is ASIC mining, and specialized hardware is available from a number of vendors such as Antminer, AvalonMiner and Whatsminer. Solo mining is not much profitable now unless a vast amount of money and energy is spent to build your own mining rig or even a data center. With the current difficulty factor (March 2018), if a user manages to produce a hash rate of 12 TH/s, they can hope to make 0.0009170 BTC (around $6) per day, which is very low as compared to the investment required to source the equipment that can produce 12 TH. Including running costs such as electricity, this turns out to be not very profitable.

For example, Antminer S9, is an efficient ASIC miner which produces hash power of 13.5 TH/s and it seems that it can produce some profit per day, which is true but a single Antminer S9 costs around 1700 GBP and combining it with electricity cost the return on investment is almost after a year's mining when it produces around 0.3 BTC. It may seem still OK, to invest but also think about the fact that the Bitcoin network difficulty keeps going up with time and during a year it will become more difficult to mine and the mining hardware will run out its utility in a few months.

Summary

We started this chapter by introducing Bitcoin and how a transaction works from a user's point of view. Then, we presented an introduction to transactions from a technical point of view. Later we discussed public and private keys that are used in Bitcoin.

In the following section, we presented addresses and its different types, following it with a discussion on transactions, its types, and usage. Next, we looked at blockchain with a detailed explanation regarding how blockchain works and what various components are included in the Bitcoin blockchain.

In the last few sections of the chapter, we presented the mining process and relevant concepts.

In the next chapter, we will examine concepts related to the Bitcoin network, its elements, and client software tools.

6
Bitcoin Network and Payments

In this chapter, we will present the Bitcoin network, relevant network protocols, and wallets. We will explore different types of wallets that are available for bitcoin. Moreover, we will examine how the Bitcoin protocol works and the types of messages exchanged on the network between nodes, during various node and network operations. Also, we will explore various advanced and modern Bitcoin protocols that have been developed to address limitations in the original Bitcoin. Finally, we'll give you an introduction to bitcoin trading and investment.
We will start with the detailed introduction of the Bitcoin network.

The Bitcoin network

The Bitcoin network is a peer-to-peer network where nodes exchange transactions and blocks. There are different types of nodes on the network. There are two main types of nodes, full nodes and SPV nodes. **Full nodes**, as the name implies, are implementations of Bitcoin core clients performing the wallet, miner, full blockchain storage, and network routing functions. However, it is not necessary to perform all these functions. **Simple Payment Verification** (**SPV**) nodes or lightweight clients perform only wallet and network routing functionality. The latest version of Bitcoin protocol is 70015 and was introduced with Bitcoin core client 0.13.2.

Some nodes prefer to be full blockchain nodes with complete blockchain as they are more secure and play a vital role in block propagation while some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function). Another type is solo miner nodes that can perform mining, store full blockchain, and act as a Bitcoin network routing node.

There are a few nonstandard but heavily used nodes that are called **pool protocol servers**. These nodes make use of alternative protocols, such as the stratum protocol. These nodes are used in mining pools. Nodes that only compute hashes use the stratum protocol to submit their solutions to the mining pool. Some nodes perform only mining functions and are called mining nodes. It is possible to run an SPV client which runs a wallet and network routing function without a blockchain. SPV clients only download the headers of the blocks while syncing with the network and when required they can request transactions from full nodes. The verification of transactions is possible by using Merkle root in the block header with Merkle branch to prove that the transaction is present in a block in the blockchain.

Most protocols on the internet are line-based, which means that each line is delimited by a carriage return (`\r`) and newline (`\n`) character. Stratum is also a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools.

The Bitcoin network is identified by its different magic values. A list is shown as follows:

Network	Magic value	Hex
main	0xD9B4BEF9	F9 BE B4 D9
testnet3	0x0709110B	0B 11 09 07

Bitcoin network magic values

Magic values are used to indicate the message origin network.

A full node performs four functions: wallet, miner, blockchain, and the network routing node.

Before we examine that how Bitcoin discovery protocol and block synchronization works, we need to understand that what are the different types of messages that Bitcoin protocol uses. The list of message types is provided here.

There are 27 types of protocol messages in total, but they're likely to increase over time as the protocol grows. The most commonly used protocol messages and their explanation are listed as follows:

- `version`: This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.

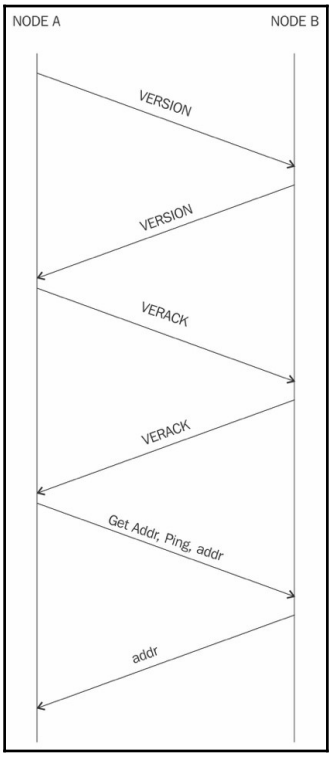
- `verack`: This is the response of the version message accepting the connection request.
- `inv`: This is used by nodes to advertise their knowledge of blocks and transactions.
- `getdata`: This is a response to `inv`, requesting a single block or transaction identified by its hash.
- `getblocks`: This returns an `inv` packet containing the list of all blocks starting after the last known hash or 500 blocks.
- `getheaders`: This is used to request block headers in a specified range.
- `tx`: This is used to send a transaction as a response to the `getdata` protocol message.
- `block`: This sends a block in response to the `getdata` protocol message.
- `headers`: This packet returns up to 2,000 block headers as a reply to the `getheaders` request.
- `getaddr`: This is sent as a request to get information about known peers.
- `addr`: This provides information about nodes on the network. It contains the number of addresses and address list in the form of IP address and port number.

When a Bitcoin core node starts up, first, it initiates the discovery of all peers. This is achieved by querying DNS seeds that are hardcoded into the Bitcoin core client and are maintained by Bitcoin community members. This lookup returns a number of DNS A records. The Bitcoin protocol works on TCP port `8333` by default for the main network and TCP `18333` for testnet. The following code shows an example of DNS seeds in `chainparams.cpp`:

```
// Pieter Wuille, only supports x1, x5, x9, and xd
vSeeds.emplace_back("seed.bitcoin.sipa.be");
// Matt Corallo, only supports x9
vSeeds.emplace_back("dnsseed.bluematt.me");
// Luke Dashjr
vSeeds.emplace_back("dnsseed.bitcoin.dashjr.org");
// Christian Decker, supports x1 - xf
vSeeds.emplace_back("seed.bitcoinstats.com");
// Jonas Schnelli, only supports x1, x5, x9, and xd
vSeeds.emplace_back("seed.bitcoin.jonasschnelli.ch");
// Peter Todd, only supports x1, x5, x9, and xd
vSeeds.emplace_back("seed.btc.petertodd.org");
```

First, the client sends a protocol message `version` that contains various fields, such as version, services, timestamp, network address, nonce, and some other fields. The remote node responds with its own `version` message followed by the `verack` message exchange between both nodes, indicating that the connection has been established.

After this, `getaddr` and `addr` messages are exchanged to find the peers that the client does not know. Meanwhile, either of the nodes can send a `ping` message to see whether the connection is still active. The `getaddr` and `addr` messages are the types defined in the Bitcoin protocol. This process is shown in the following protocol diagram:



Visualization of node discovery protocol

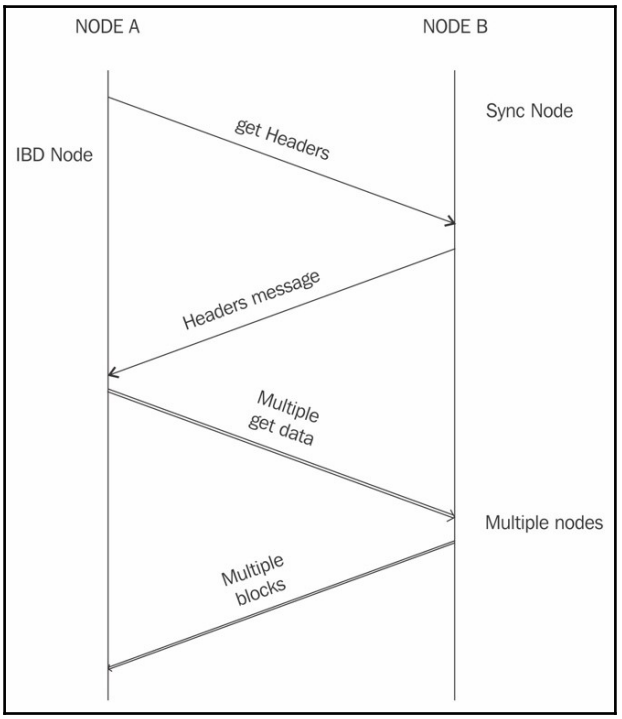
The preceding network protocol sequence diagram shows communication between two Bitcoin nodes during initial connectivity. **NODE A** is shown on the left side and **NODE B** on the right. First, **NODE A** starts the connection by sending the `version` message which contains version number and current time to the remote peer **NODE B**. **NODE B** then responds with its own `version` message containing the version number and current time. **NODE A** and **NODE B** then exchange a `verack` message indicating that the connection has been successfully established. After the connection is successful the peers can exchange `getaddr` and `addr` messages to discover other peers on the network.

Now the block download can begin. If the node already has all the blocks fully synchronized, then it listens for new blocks using the `inv` protocol message; otherwise, it first checks whether it has a response to `inv` messages and have inventories already. If yes, then it requests the blocks using the `getdata` protocol message; if not, then it requests inventories using the `getblocks` message. This method was used until version 0.9.3. This was a slower process known as **blocks-first approach** and was replaced with **headers-first approach** in 0.10.0.

Initial block download can use blocks-first or the headers-first method to synchronize blocks depending on the version of the Bitcoin core client. The blocks-first method is very slow and was discontinued since February 16, 2015 with the release of version 0.10.0.

Since version 0.10.0, the initial block download method named headers-first was introduced. This resulted in major performance improvement and the blockchain synchronization that used to take days to complete started taking only a few hours. The core idea is that the new node first asks peers for block headers and validates them. Once this is completed, blocks are requested in parallel from all available peers as the blueprint of the complete chain is already downloaded in the form of the block header chain.

In this method, when the client starts up, it checks whether the blockchain is fully synchronized already if the header chain is already synchronized; if not, which is the case the first time the client starts up, it requests headers from other peers using the `getheaders` message. If the blockchain is fully synchronized, it listens for new blocks via `inv` messages, and if it already has a fully synchronized header chain, then it requests blocks using the `getdata` protocol messages. The node also checks whether the header chain has more headers than blocks and then it requests blocks by issuing the `getdata` protocol message.



Bitcoin core client >= 0.10.0 header and block synchronization

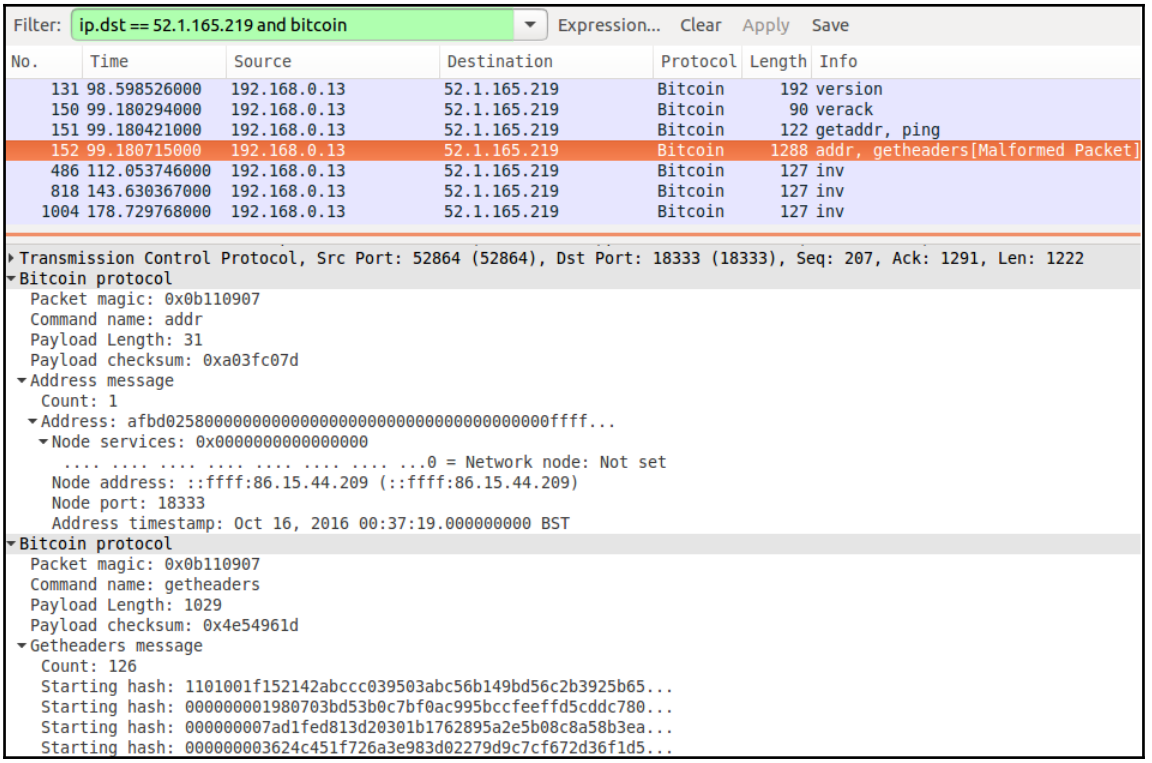
The preceding diagram shows the Bitcoin block synchronization process between two nodes on the Bitcoin network. **NODE A**, shown on the left side is called **Initial Block Download Node** (**IBD Node**) and **NODE B**, shown on the right is called **Sync Node**.

IBD Node means that this is the node that is requesting the blocks and **Sync Node** means the node from where the blocks are being requested. The process starts by **NODE A** first sending the `getheaders` message which is responded with the `headers` message from the **Sync Node**. The payload of the `getheaders` message is one or more header hashes. If it's a new node then there is only the first genesis block's header hash. The **Sync Node** replies with sending up to 2,000 block headers to the **IBD Node**. After this, the **IBD Node** starts to download more headers from the **Sync Node** and in parallel, downloads blocks from multiple nodes. In other words, **IBD Node** makes requests to multiple nodes and as a result multiple blocks are sent to **IBD Node** from **Sync Node** and other nodes. If the **Sync Node** does not have more headers than 2,000, when **IBD Node** makes a `getheaders` request then **IBD Node** sends `getheaders` message to other nodes. This process continues in parallel until the blockchain synchronization is complete.

The `getblockchaininfo` and `getpeerinfo` RPCs were updated with a new functionality to cater for this change. A **Remote Procedure Call** (**RPC**), `getchaintips`, is used to list all known branches of the blockchain. This also includes headers only blocks. The `getblockchaininfo` RPC is used to provide the information about the current state of the blockchain. The `getpeerinfo` RPC is used to list both the number of blocks and the headers that are in common between peers.

Wireshark can also be used to visualize message exchange between peers and can serve as an invaluable tool to learn about the Bitcoin protocol. A sample is shown here. This is a basic example showing the `version`, `verack`, `getaddr`, `ping`, `addr`, and `inv` messages.

In the details, valuable information such as the packet type, command name, and results of the protocol messages can be seen:



A sample block message in Wireshark

A protocol graph showing the flow of data between the two peers is shown in the preceding diagram. This can help you understand when a node starts up and what type of messages are used.

In the following example, the Bitcoin dissector is used to analyze the traffic and identify the Bitcoin protocol commands.

Exchange of messages such as `version`, `getaddr`, and `getdata` can be seen in the following example along with the appropriate comment describing the message name.

This exercise can be very useful in order to learn Bitcoin protocol and it is recommended that the experiments be carried out on the Bitcoin testnet (`https://en.bitcoin.it/wiki/Testnet`), where various messages and transactions can be sent over the network and then be analyzed by Wireshark.

Wireshark is a network analysis tool available at `https://www.wireshark.org`.

The analysis performed by Wireshark in the following screenshot shows the exchange of messages between two nodes. If you look closely, you'll notice that top three messages show the node discovery protocol that we have discussed before:

Time	192.168.0.13 (57868) ⟷ 136.243.139.96 (18333)	Comment
97.734135000	version	Bitcoin: version
98.025045000	verack	Bitcoin: verack
98.025177000	getaddr, pin...	Bitcoin: getaddr, ping, addr
98.025468000	getheaders, ...	Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers
98.160419000	[TCP Retran...	Bitcoin: [TCP Retransmission] , getheaders, [unknown command], [unknown command], [unknown command]
98.598399000	getdata	Bitcoin: getdata
144.343544000	inv	Bitcoin: inv
176.152240000	getdata	Bitcoin: getdata
179.493755000	getdata	Bitcoin: getdata
218.101646000	ping	Bitcoin: ping
218.192004000	[unknown co...	Bitcoin: [unknown command]
218.444431000	[TCP Retran...	Bitcoin: [TCP Retransmission] , [unknown command]
336.234936000	getdata	Bitcoin: getdata
337.843423000	[unknown co...	Bitcoin: [unknown command]
338.143885000	ping	Bitcoin: ping
448.764093000	getdata	Bitcoin: getdata
457.894823000	[unknown co...	Bitcoin: [unknown command]
458.195265000	ping	Bitcoin: ping
578.011774000	[unknown co...	Bitcoin: [unknown command]
578.212044000	ping	Bitcoin: ping
585.587671000	inv	Bitcoin: inv
647.169633000	inv	Bitcoin: inv
671.962545000	getdata	Bitcoin: getdata
698.037067000	[unknown co...	Bitcoin: [unknown command]
698.237350000	ping	Bitcoin: ping
701.563581000	inv	Bitcoin: inv
701.986269000	inv	Bitcoin: inv
705.022173000	inv	Bitcoin: inv
812.115878000	inv	Bitcoin: inv
818.198570000	[unknown co...	Bitcoin: [unknown command]
818.298733000	ping	Bitcoin: ping

Node discovery protocol in Wireshark

Full clients are thick clients or full nodes that download the entire blockchain; this is the most secure method of validating the blockchain as a client. Bitcoin network nodes can operate in two fundamental modes: full client or lightweight SPV client. SPV clients are used to verify payments without requiring the download of a full blockchain. SPV nodes only keep a copy of block headers of the current valid longest blockchain. Verification is performed by looking at the Merkle branch that links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more practical approach, which was implemented with BIP 37 (`https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`), where bloom filters were used to filter out relevant transactions only.

Bloom filter is basically a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It basically provides probabilistic lookup with false positives but no false negatives. It means that this filter can produce an output where an element that is not a member of the set being tested is wrongly considered to be in the set, but it can never produce an output where an element does exist in the set, but it asserts that it does not.

Elements are added to the bloom filter after hashing them several times and then set the corresponding bits in the bit vector to `1` via the corresponding index. In order to check the presence of the element in the bloom filter, the same hash functions are applied and compared with the bits in the bit vector to see whether the same bits are set to `1`. Not every hash function (such as SHA-1) is suitable for bloom filters as they need to be fast, independent, and uniformly distributed. The most commonly used hash functions for bloom filters are FNV, Murmur, and Jenkins.

These filters are mainly used by SPV clients to request transactions and the Merkle blocks they are interested in. A Merkle block is a lightweight version of the block, which includes a block header, some hashes, a list of 1-bit flags, and a transaction count. This information can then be used to build a Merkle tree. This is achieved by creating a filter that matches only those transaction and blocks that have been requested by the SPV client. Once the `version` messages have been exchanged and connection has been established between peers, the nodes can set filters according to their requirements.

These probabilistic filters offer a varying degree of privacy or precision depending upon how accurately or loosely they have been set. A strict bloom filter will only filter transactions that have been requested by the node but at the expense of the possibility of revealing the user addresses to adversaries who can correlate transactions with their IP addresses, thus compromising privacy. On the other hand, a loosely set filter can result in retrieving more unrelated transactions but will offer more privacy. Also, for SPV clients, bloom filters allow them to use low bandwidth as opposed to downloading all transactions for verification.

BIP 37 proposed the Bitcoin implementation of bloom filters and introduced three new messages to the Bitcoin protocol:

- `filterload`: This is used to set the bloom filter on the connection
- `filteradd`: This adds a new data element to the current filter
- `filterclear`: This deletes the currently loaded filter

More details can be found in the BIP 37 specification. This is available at `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

Wallets

The wallet software is used to store private or public keys and Bitcoin address. It performs various functions, such as receiving and sending bitcoins. Nowadays, software usually offers both functionalities: Bitcoin client and wallet. On the disk, the Bitcoin core client wallets are stored as the Berkeley DB file:

```
$ file wallet.dat
wallet.dat: Berkeley DB (B-tree, version 9, native byte-order)
```

Private keys are generated by randomly choosing a 256-bit number by wallet software. The rules of generation are predefined and were discussed in `Chapter 4`, *Public Key Cryptography*. Private keys are used by wallets to sign the outgoing transactions. Wallets do not store any coins, and there is no concept of wallets storing balance or coins for a user. In fact, in the Bitcoin network, coins do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the number of bitcoins.

In Bitcoin, there are different types of wallets that can be used to store private keys. As a software program, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network.

Non-deterministic wallets

These wallets contain randomly generated private keys and are also called *just a bunch of key wallets*. The Bitcoin core client generates some keys when first started and generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process can lead to theft and loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them in order to prevent theft or loss.

Deterministic wallets

In this type of wallet, keys are derived out of a seed value via hash functions. This seed number is generated randomly and is commonly represented by human-readable *mnemonic code* words. Mnemonic code words are defined in BIP 39, a Bitcoin improvement proposal for mnemonic code for generating deterministic keys. This BIP is available at `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`. This phrase can be used to recover all keys and makes private key management comparatively easier.

Hierarchical Deterministic wallets

Defined in BIP32 and BIP44, **Hierarchical Deterministic** (**HD**) wallets store keys in a tree structure derived from a seed. The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys. The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known. It is because of this property that HD wallets are very easy to maintain and are highly portable. There are many free and commercially available HD wallets available. For example, Trezor (`https://trezor.io`), Jaxx (`https://jaxx.io/`) and Electrum (`https://electrum.org/`).

Brain wallets

The master private key can also be derived from the hash of passwords that are memorized. The key idea is that this passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as a brain wallet. This method is prone to password guessing and brute force attacks but techniques such as key stretching can be used to slow down the progress made by the attacker.

Paper wallets

As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored.

Paper wallets can be generated online from various service providers, such as `https://bitcoinpaperwallet.com/` or `https://www.bitaddress.org/`.

Hardware wallets

Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built or with the advent of NFC-enabled phones, this can also be a **Secure Element** (**SE**) in NFC phones. Trezor and Ledger wallets (various types) are the most commonly used Bitcoin hardware wallets. The following is the photo of a Trezor wallet:

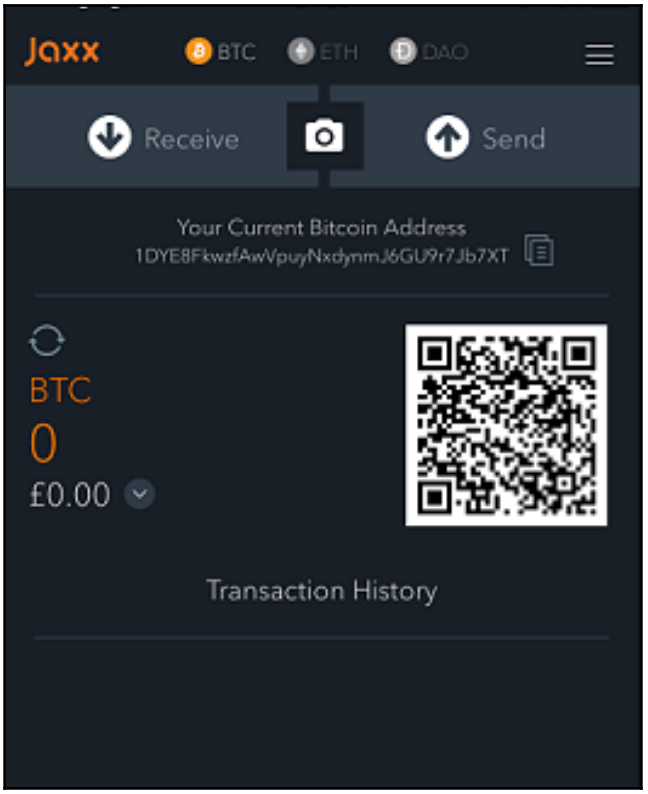
Trezor wallet

Online wallets

Online wallets, as the name implies, are stored entirely online and are provided as a service usually via the cloud. They provide a web interface to the users to manage their wallets and perform various functions such as making and receiving payments. They are easy to use but imply that the user trusts the online wallet service provider. An example of online wallet is GreenAddress, which is available at `https://greenaddress.it/en/`.

Mobile wallets

Mobile wallets, as the name suggests, are installed on mobile devices. They can provide various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments. Mobile wallets are available for the Android platform and iOS, for example, Blockchain, breadwallet, Copay, and Jaxx.



Jaxx mobile wallet

The choice of Bitcoin wallet depends on several factors such as security, ease of use, and available features. Out of all these attributes, security of course comes first and when making a decision about which wallet to use, security should be of paramount importance. Hardware wallets tend to be more secure as compared to web wallets because of their tamper resistant design. Web wallets by nature are hosted on websites, which may not be as secure as a tamper resistant hardware device. Generally, mobile wallets for smartphone devices are quite popular due to a balanced combination of features and security. There are many companies offering these wallets on the iOS App Store and Android Play. It is however quite difficult to suggest that which type of wallet should be used, it also depends on personal preferences and features available in a wallet. It is advisable that security should be kept in mind while making decision on which wallet to choose.

Bitcoin payments

Bitcoins can be accepted as payments using various techniques. Bitcoin is not recognized as a legal currency in many jurisdictions, but it is increasingly being accepted as a payment method by many online merchants and e-commerce websites. There are a number of ways in which buyers can pay the business that accepts bitcoins. For example, in an online shop, Bitcoin merchant solutions can be used, whereas in traditional, physical shops, point of sale terminals and other specialized hardware can be used. Customers can simply scan the QR code with the seller's payment URI in it and pay using their mobile devices. Bitcoin URIs allow users to make payments by simply clicking on links or scanning QR codes. **Uniform Resource Identifier** (**URI**) is basically a string that represents the transaction information. It is defined in BIP 21. The QR code can be displayed near the point of the sale terminal. Nearly all Bitcoin wallets support this feature.

Businesses can use the following logo to advertise that they accept bitcoins as payment from customers.

bitcoin accepted here logo

Various payment solutions, such as XBTerminal and 34 Bytes bitcoin **Point of Sale** (**POS**) terminal are available commercially.

Generally, these solutions work by following these steps:

1. The sales person enters the amount of money to be charged in Fiat currency, for example, US Dollars
2. Once the value is entered in the system the terminal prints a receipt with QR code on it and other relevant information such as amount
3. The customer can then scan this QR code using their mobile Bitcoin wallet to send the payment to the Bitcoin address of the seller embedded within the QR code
4. Once the payment is received on the designated Bitcoin address, a receipt is printed out as a physical evidence of sale

A Bitcoin POS device from 34 Bytes is shown here:

34 Bytes POS solution

The bitcoin payment processor, offered by many online service providers, allows integration with e-commerce websites. There are many options available. These payment processors can be used to accept bitcoins as payments. Some service providers also allow secure storage of bitcoins. For example, bitpay, `https://bitpay.com`. Another example is Bitcoin Merchant Solutions available at `https://www.bitcoin.com/merchant-solutions`.

Various **Bitcoin Improvement Proposals** (**BIPs**) have been proposed and finalized in order to introduce and standardize bitcoin payments. Most notably, BIP 70 (*Payment Protocol*) describes the protocol for secure communication between a merchant and customers. This protocol uses X.509 certificates for authentication and runs over HTTP and HTTPS. There are three messages in this protocol: `PaymentRequest`, `Payment`, and `PaymentACK`. The key features of this proposal are defense against man-in-the-middle attacks and secure proof of payment. Man-in-the-middle attacks can result in a scenario where the attacker is sitting between the merchant and the buyer and it would seem to the buyer that they are talking to the merchant, but in fact, the man in the middle is interacting with the buyer instead of the merchant. This can result in manipulation of the merchant's Bitcoin address to defraud the buyer.

Several other BIPs, such as BIP 71 (*Payment Protocol MIME types*) and BIP 72 (*URI extensions for Payment Protocol*), have also been implemented to standardize payment scheme to support BIP 70 (*Payment Protocol*).

Bitcoin lightning network, is a solution for scalable off-chain instant payments. It was introduced in early 2016, which allows off-blockchain payments. This network makes use of payments channels that run off the blockchain which allows greater speed and scalability of Bitcoin.

This paper is available at `https://lightning.network` and interested readers are encouraged to read the paper in order to understand the theory and rationale behind this invention.

Innovation in Bitcoin

Bitcoin has undergone many changes and still evolving into a more and more robust and better system by addressing various weaknesses in the system. Especially, performance has been a topic of hot debate among Bitcoin experts and enthusiasts for many years. As such, various proposals have been made in the last few years to improve Bitcoin performance resulting in greater transaction speed, increased security, payment standardization and overall performance improvement at the protocol level.

These improvement proposals are usually made in the form of BIPs or fundamentally new versions of Bitcoin protocols resulting in a new network altogether. Some of the changes proposed are implementable via a soft fork but few need a hard fork and as a result, give birth to a new currency.

In the following sections, we will see what are the various BIPs that can be proposed for improvement in Bitcoin and then we will discuss some advanced protocols that have been proposed and implemented to address various weaknesses in the Bitcoin.

Bitcoin Improvement Proposals (BIPs)

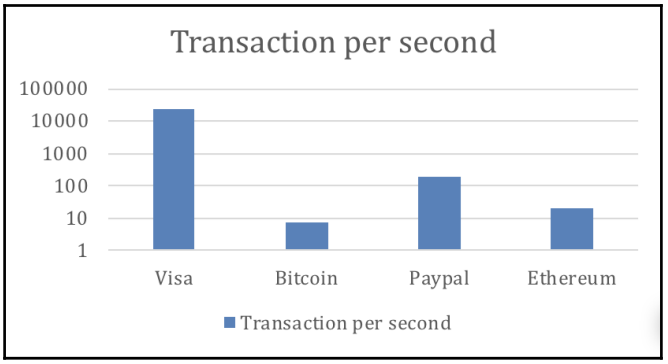
These documents are used to propose or inform the Bitcoin community about the improvements suggested, the design issues, or information about some aspects of the bitcoin ecosystem. There are three types of Bitcoin improvement proposals, abbreviated as BIPs:

- **Standard BIP**: Used to describe the major changes that have a major impact on the Bitcoin system, for example, block size changes, network protocol changes, or transaction verification changes.
- **Process BIP**: A major difference between standard and process BIPs is that standard BIPs cover protocol changes, whereas process BIPs usually deal with proposing a change in a process that is outside the core Bitcoin protocol. These are implemented only after a consensus among bitcoin users.
- **Informational BIP**: These are usually used to just advise or record some information about the Bitcoin ecosystem, such as design issues.

Advanced protocols

In this section, we will see that what are the various advanced protocols that have been suggested or implemented for improving the Bitcoin protocol.

Transaction throughput is one of the critical issues that need to be addressed. Inherently, the Bitcoin network can only process from approximately 3 to 7 transactions per second which is a tiny number as compared to other financial networks, such as Visa which can process approximately, on average, 24,000 transactions per second. PayPal can process approximately 200 transactions per second whereas Ethereum can process up to on average 20. As Bitcoin Network grew exponentially over the last few years, these issues started to grow even further. The difference of processing speed is also shown below in a graph which shows the scale of difference between Bitcoin and other networks' transaction speeds.



Bitcoin transaction speed as compared to other networks (on logarithmic scale)

Also, security issues such as transaction malleability are of real concern which can result in denial of service. Various proposals have been made to improve the Bitcoin proposal to address various weaknesses. A selection of these proposals is presented in the following subsections.

Segregated Witness (SegWit)

The SegWit or Segregated Witness is a soft fork based update to the Bitcoin protocol which addresses some weaknesses such as throughput and security in the Bitcoin protocol. SegWit offers a number of improvements as listed here:

- Fix for transaction malleability due to the separation of signature data from transactional data. In this case, it is no longer possible to modify transaction ID because it is no longer calculated based on the signature data present within the transaction.
- Reduction in transaction size results in cheaper transaction fees.
- Reduction in transaction signing and verification times, which results in faster transactions.
- Script versioning, which allows version number to be prefixed to locking scripts. This change can result in improvements in the scripting language without requiring a hard fork and by just increasing the version number of the script.
- Reduction in input verification time.

SegWit was proposed in BIP 141, BIP 143, BIP 144 and BIP 145. It has been activated on Bitcoin main network on August 24, 2017. The key idea behind SegWit is the separation of signature data from transaction data, which results in reduced size of the transaction. This results in block size increase up to 4 MB. However, the practical limit is between 1.6 MB to 2 MB. Instead of hard size limit of 1 MB blocks, SegWit has introduced a new concept of block weight limit.

To spend an **Unspent Transaction Output** (**UTXO**) in Bitcoin, it needs a valid signature to be provided. In the pre-SegWit scenario, this signature is provided within the locking script whereas in SegWit this signature is not part of the transaction and is provided separately.

There are two types of transaction that can now be constructed using SegWit wallets but note that these are not new transaction types as such, these are just new ways by which UTXOs can be spent. These types are:

- **Pay to Witness public key hash** (**P2WPKH**)
- **Pay to Witness Script hash** (**P2WSH**)

Bitcoin Cash

Bitcoin Cash increases the block limit to 8 MB. This immediately increases the number of transactions that can be processed in one block to a much larger number as compared to 1 MB limit in original Bitcoin protocol. It uses PoW as consensus algorithm, and mining hardware is still ASIC based. The block interval is changed from 10 minutes to 10 seconds and up to 2 hours. It also provides replay protection and wipe-out protection.

Bitcoin Unlimited

In this proposal, the size of the block is increased but not set to a hard limit. Instead, miners come to a consensus on the block size cap over a period of time. Other concepts such as *parallel validation* and *extreme thin blocks* have also been proposed in Bitcoin Unlimited.

Its client is available for download at `https://www.bitcoinunlimited.info`.

Extreme thin blocks allow for a faster block propagation between Bitcoin nodes. In this scheme the node requesting blocks sends a `getdata` request along with a bloom filter to another node. Purpose of this bloom filter is to filter out the transactions that already exists in the **mempool** (short for **memory pool**) of the requesting node. The node then sends back a *thin block* only containing the missing transactions. This fixes an inefficiency in Bitcoin where by transaction are regularly received twice, once at the time of broadcast by the sender and then again when a mined block is broadcasted with the confirmed transaction.

Parallel validation allows nodes to validate more than one block along with new incoming transactions in parallel. This mechanism is in contrast to Bitcoin where a node during its validation period after receiving a new block cannot relay new transactions or validate any blocks until it has accepted or rejected the block.

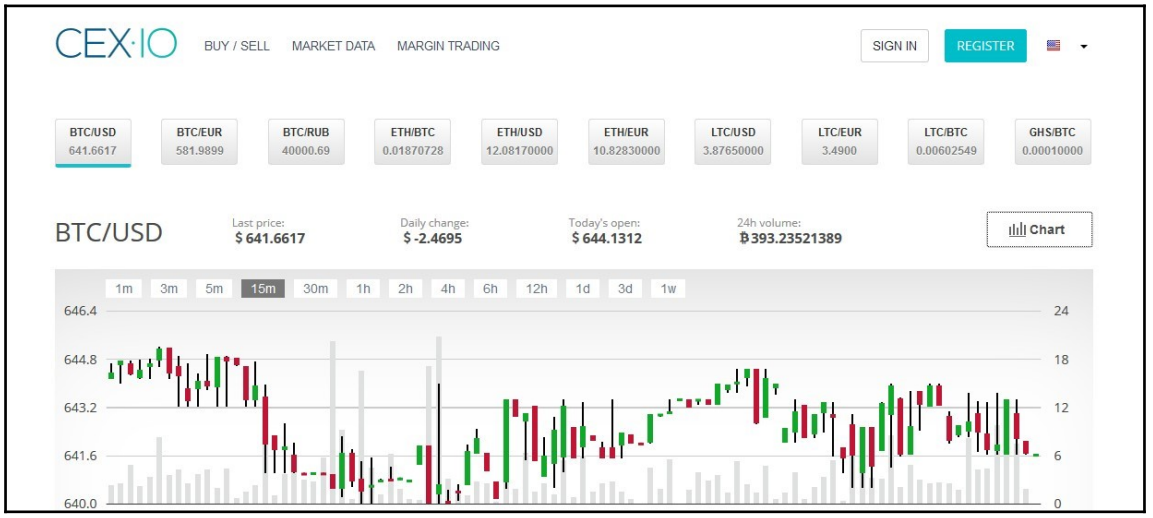
Bitcoin Gold

This proposal has been implemented as a hard fork since block 49,1407 of the original Bitcoin blockchain. Being a hard fork, it resulted in a new blockchain, named **Bitcoin Gold** (**BTG**). The core idea behind this concept is to address the issue of mining centralization which has hurt the original Bitcoin idea of decentralized digital cash whereby more hash power has resulted in a power shift towards miners with more hashing power. BTG uses the Equihash algorithm as its mining algorithm instead of PoW; hence it is inherently ASIC resistant and uses GPUs for mining.

There are other proposals like Bitcoin Next Generation, Solidus, Spectre, and SegWit2x which will be discussed later in this book in `Chapter 18`, *Scalability and Other Challenges*, in the context of performance improvement in blockchain networks.

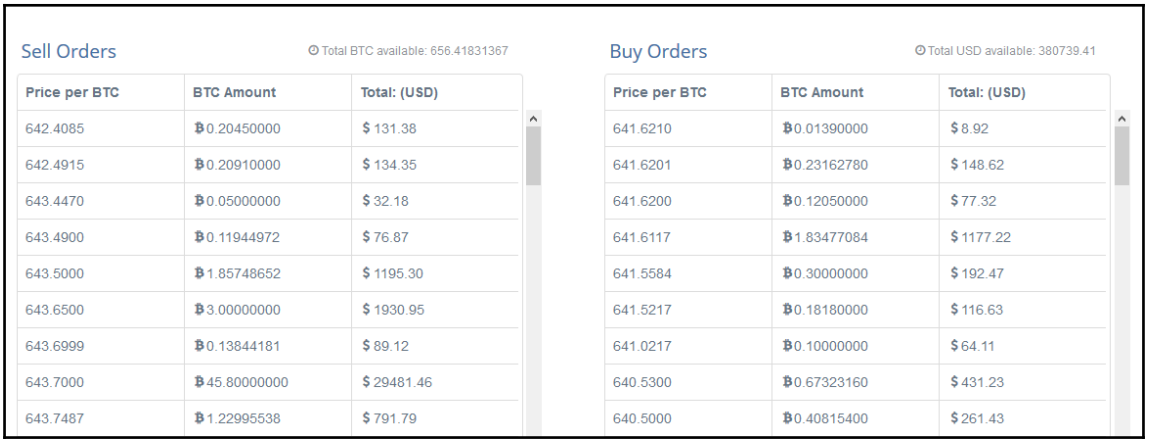
Bitcoin investment and buying and selling bitcoins

There are many online exchanges where users can buy and sell bitcoins. This is a big business on the internet now and it offers bitcoin trading, CFDs, spread betting, margin trading, and various other choices. Traders can buy bitcoins or trade by opening long or short positions to make a profit when bitcoin's price goes up or down. Several other features, such as exchanging bitcoins for other virtual currencies, are also possible, and many online bitcoin exchanges provide this function. Advanced market data, trading strategies, charts, and relevant data to support traders is also available. An example is shown from CEX (`https://cex.io`) here. Other exchanges offer similar types of services.



Example of bitcoin exchange cex.io

The following screenshot shows the order book at the exchange where all buy and sell orders are listed:



Sell Orders — Total BTC available: 656.41831367

Price per BTC	BTC Amount	Total: (USD)
642.4085	Ƀ0.20450000	$ 131.38
642.4915	Ƀ0.20910000	$ 134.35
643.4470	Ƀ0.05000000	$ 32.18
643.4900	Ƀ0.11944972	$ 76.87
643.5000	Ƀ1.85748652	$ 1195.30
643.6500	Ƀ3.00000000	$ 1930.95
643.6999	Ƀ0.13844181	$ 89.12
643.7000	Ƀ45.80000000	$ 29481.46
643.7487	Ƀ1.22995538	$ 791.79

Buy Orders — Total USD available: 380739.41

Price per BTC	BTC Amount	Total: (USD)
641.6210	Ƀ0.01390000	$ 8.92
641.6201	Ƀ0.23162780	$ 148.62
641.6200	Ƀ0.12050000	$ 77.32
641.6117	Ƀ1.83477084	$ 1177.22
641.5584	Ƀ0.30000000	$ 192.47
641.5217	Ƀ0.18180000	$ 116.63
641.0217	Ƀ0.10000000	$ 64.11
640.5300	Ƀ0.67323160	$ 431.23
640.5000	Ƀ0.40815400	$ 261.43

Example of bitcoin order book at exchange cex.io

The order book shown in the preceding screenshot displays sell and buy orders. Sell orders are also called ask and buy orders are also called bid orders. This means that ask price is at what seller is willing to sell the bitcoin whereas bid price is what the buyer is willing to pay. If bid and ask prices match then a trade can occur. Most common order types are market orders and limit orders. Market orders mean that as soon as the prices match the order will be filled immediately. Limit orders allow buying and selling of set number of bitcoins at a specified price or better. Also, a period of time can be set during which the order can be left open, if not executed then it will be cancelled. We have introduced trading concepts in more detail in `Chapter 4`, *Public Key Cryptography*, under *Financial markets* section, interested readers can refer to this section for more details.

Summary

We started this chapter with the introduction to Bitcoin network, following it with a discussion on Bitcoin node discovery and block synchronization protocols. Moreover, we presented different types of network messages. Then we examined different types of Bitcoin wallets and discussed various attributes and features of each type. Following this, we looked at Bitcoin payments and payment processors. In the last section, we discussed Bitcoin innovations, which included topics such as Bitcoin Improvement Proposals, and advanced Bitcoin protocols. Finally, we presented a basic introduction to Bitcoin buying and selling.

In the next chapter, we will discuss Bitcoin clients, such as Bitcoin Core client, which can be used to interact with the Bitcoin blockchain and also acts as a wallet. In addition, we will explore some of the APIs that are available for programming Bitcoin applications.