3
Cryptography and Technical Foundations

In this chapter, you will be introduced to the concepts, theory, and practical aspects of cryptography. More focus will be given to aspects that are specifically relevant in the context of the blockchain technology. Moreover, concepts from financial markets will also be discussed in order to provide a basis for the material covered in later chapters.

You will also be introduced to the practical implementations of cryptographic algorithms so that you can experience the cryptographic functions practically. For this, the **OpenSSL** command line is used.

Before starting the theoretical foundations, the installation of OpenSSL is discussed in the following section so that you can do some practical work as you read through the theoretical material.

On Ubuntu Linux distribution, OpenSSL is usually already available; however, it can be installed using the following commands:

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

In the upcoming sections, first, the theoretical foundation will be discussed and then relevant practical experiments will be introduced.

Introduction

Cryptography is the science of making information secure in the presence of adversaries. It provides a means of secure communication in the presence of adversaries with assumed limitless resources. Ciphers are used to encrypt data so that if intercepted by an adversary, the data is meaningless to them without decryption, which requires the secret key.

Cryptography is generally used to provide a confidentiality service. On its own, it cannot be considered a complete solution but serve as a crucial building block within a larger security system to address a security problem.

Cryptography provides various security services, such as **Confidentiality, Integrity, Authentication**, (Entity Authentication and Data origin authentication) and **non-repudiation**. Additionally, accountability is also required in various security systems.

Before discussing cryptography further, there are some mathematical terms and concepts that need to be explained first in order to fully understand the material provided later in this chapter. The next section introduces these concepts. It should be noted that this section is intended as a basic introduction. An explanation with proofs and relevant background for all these terms will require rather 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 text book.

Mathematics

As the subject of cryptography is based on mathematics, this section will introduce some basic concepts that will help you understand the concepts 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 an identity element defined. 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 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 terms or factors are reordered.

A finite field

A finite field is a field 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 to construct discrete logarithm problem.

Order

This is the number of elements in a field. It is also known as the cardinality of the field.

Prime fields

This is a finite field 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.

Ring

If more than one operation can be defined over an abelian group, that group becomes a ring. There are also certain 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. In other words, if the group operation is repeatedly applied to a particular element in the group, then all elements in the group can be generated.

An abelian group

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

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 with regard to this fixed number. In an analogy to a clock, there are number from 1 to 12. When it reaches 12, the number 1 starts again. In other words, this 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; in the next section, you will be introduced to cryptography.

Cryptography

As discussed earlier, cryptography provides various security services, and these security services are discussed here.

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 authentications, discussed here.

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, which are used to gain access to the platforms they are using. This is called single factor authentication as there is only one factor, namely *something you know*, that is, the password and username. This type of authentication is not very secure due to various reasons, such as 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. If more than two factors are used for authentication, that is called multifactor authentication. Various factors are described here:

1. The first factor is something you have, such as a hardware token or smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. This provides protection by requiring two factors of authentication. A user who has access to the hardware token and knows the log-on credentials will be able to access the system. Both factors should be available in order to gain access to the system, thus making this method a two-factor authentication mechanism.

2. The second factor is something you are, which uses biometric features in order to identify the user. In this method, a user uses fingerprint, retina, iris, or hand geometry to provide an additional factor for authentication. This way, it can be ensured that a user was indeed present during the authentication mechanism as biometric features are unique to an individual. However, careful implementation is required in order to ensure a high level of security as some research has suggested that biometric systems can be circumvented in certain scenarios.

Data origin authentication

Also known as message authentication, this is an assurance that the source of information is verified. Data origin authentication implies data integrity because if a source is corroborated, then 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 unforgeable evidence. It is a security service that provides unforgeable evidence that a particular action has occurred. This property is very necessary in disputable situations whereby an entity has denied actions performed, for example, placing 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 in order to increase the confidence level of consumers in the service.

The non-repudiation protocol usually runs in a communication network and 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 communication models that can be used to transfer messages from originator *A* to recipient *B*:

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

The main 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 nonrepudiation protocols** (**MPNR**) has been developed.

Accountability

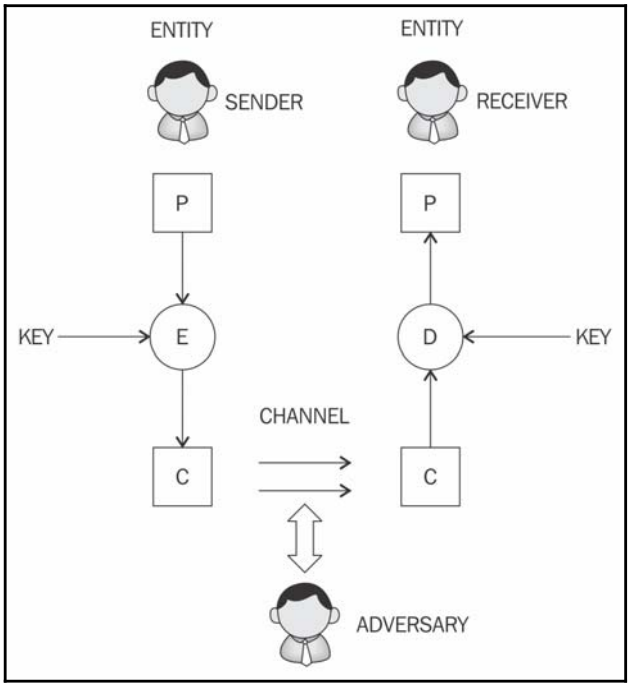
Accountability is the assurance that actions affecting security can be traced to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are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

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 the building of secure protocols and systems. A **security protocol** is a set of steps taken in order to achieve 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**.

A generic cryptography model is shown in the following diagram:



A model showing the generic encryption and decryption model

In the preceding diagram, **P**, **E**, **C**, and **D** represents Plain text, Encryption, Cipher text, and Decryption, respectively. Also, based on the model shown earlier, it is worth explaining various concepts such as entity, sender, receiver, adversary, key, and a channel.

- **Entity**: It is either a person or a system that sends, receives, or performs operations on data
- **Sender**: Sender is an entity that transmits the data
- **Receiver**: Receiver 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 some data that is used to encrypt or decrypt data
- **Channel**: Channel provides a medium of communication between entities

Cryptography is mainly divided into two categories, namely symmetric and asymmetric cryptography.

Symmetric cryptography

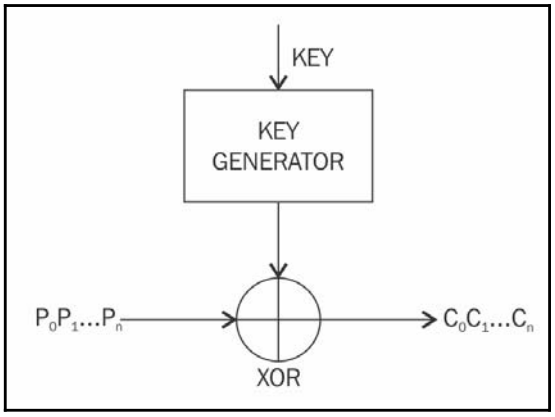
Symmetric cryptography refers to a type of cryptography whereby the key that is used to encrypt the data is the same for decrypting the data, and thus it is also known as a shared key cryptography. The key must be established or agreed on before the data exchange between the communicating parties. This 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 common examples of block ciphers, whereas RC4 and A5 are commonly used stream ciphers.

Stream ciphers

These ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis to plain text using a key stream. There are two types of stream ciphers: synchronous and asynchronous. Synchronous stream ciphers are ones where key stream is dependent only on the key, whereas asynchronous stream ciphers have a key stream that is also dependent on the encrypted data.

In stream ciphers, encryption and decryption are basically the same function because they are simple modulo 2 additions or XOR operation. The key requirement in stream ciphers is the security and randomness of key streams. Various techniques have been developed to generate random numbers, and it's vital that all key generators be cryptographically secure:



Operation of a stream cipher

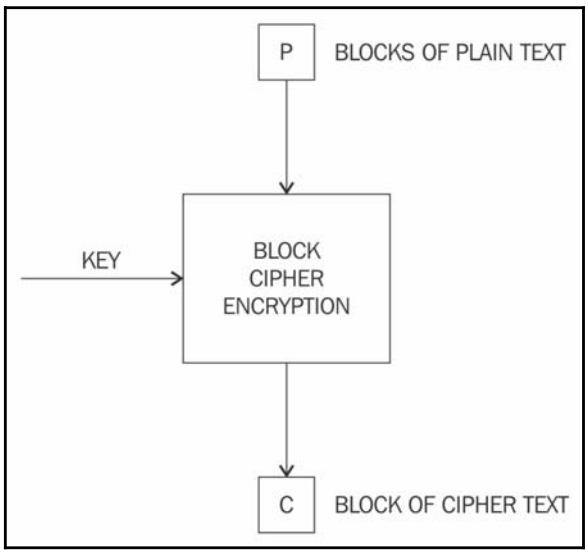
Block ciphers

These are encryption algorithms that break up a text to be encrypted (plain text) into blocks of fixed length and apply encryption block by block. Block ciphers are usually built using a design strategy known as Fiestel cipher. Recent block ciphers, such as AES (Rijndael) have been built using a combination of substitution and permutation called **substitution-permutation network** (**SPN**).

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

Confusion makes the relationship between the encrypted text and plaintext complex. This is achieved by substitution in practice. For example, 'A' in plain text 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 plain text statistically over the encrypted data, which 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 cipher text. 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 Fiestel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process in order to achieve decryption. DES is a prime example of Fiestel-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 Mode** (**OFB**), or **Counter mode** (**CTR**). These modes are used to specify the way in which an encryption function would be applied to the plain text. These modes will be explained later in this section, but the first four categories of block cipher encryption modes are introduced here.

Block encryption mode

In this mode, plaintext is divided into blocks of fixed length depending on the type of cipher used and then the encryption function is applied on each block.

Keystream generation modes

In this mode, the encryption function generates a keystream that is then XORed with the plaintext stream in order to achieve encryption.

Message authentication modes

In this mode, a message authentication code is computed as a result of an encryption function. MAC is basically a cryptographic checksum that provides an integrity service. The most common method to generate MAC using block ciphers is CBC-MAC, where some part of the last block of the chain is used as a MAC.

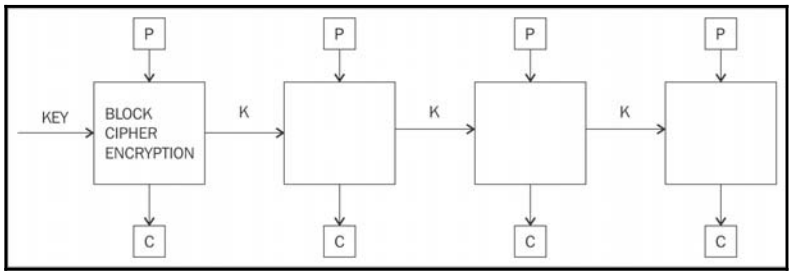
Cryptographic hashes

Hash functions are basically used to compress a message to a fixed length digest. In this mode, block ciphers are used as a compression function to produce a hash of plain text.

The most common block encryption modes are discussed briefly.

Electronic code book

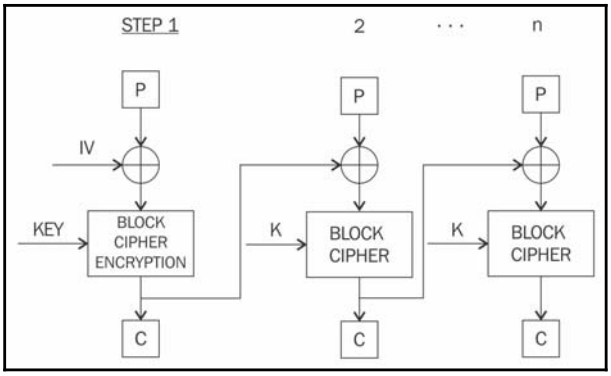
This is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm one by one separately to each block of plain text. This is the simplest mode but should not be used in practice as it is insecure and can reveal information:



Electronic code book mode for block ciphers

Cipher block chaining

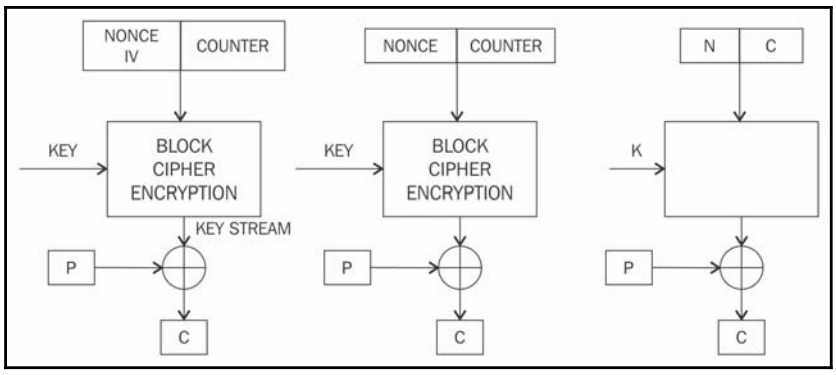
In this mode, each block of plain text is XORed with the previous encrypted block. The CBC mode uses initialization vector IV to encrypt the first block. It is recommended that IV be randomly chosen:



Cipher block chaining mode

Counter mode

The 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 in order to produce a **key stream**:



Counter mode

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

In the following section, you will be introduced to the design and mechanism of a currently dominant block cipher know as AES. First, some history will be presented with regard to Data Encryption Standard (DES) that led to the development of a new AES standard.

Data Encryption Standard (DES)

DES was introduced by the US **National Institute of Standards and Technology** (**NIST**) as a standard algorithm for encryption and was in main use during 1980s and 1990s, but it has been not proven to be very resistant against brute force attacks, due to advances in technology and cryptography research. Especially in July 1998, **Electronic Frontier Foundation** (**EFF**) broke DES using a special purpose machine. DES uses a key of only 56 bits, which has raised some concerns. This problem was addressed with the introduction of **Triple DES** (**3DES**), which proposed the usage of a 168-bit key using three 56-bit keys and the same number of executions of the DES algorithm, thus making brute force attacks almost impossible. But other limitations, such as slow performance and 64-bit block size, are not desirable.

Advanced Encryption Standard (AES)

In 2001, after an open competition, an encryption algorithm named Rijndael that was invented by cryptographers *Joan Daemen* and *Vincent Rijmen* was standardized as AES with minor modifications by NIST in 2001. So far, no attack has been found against AES that is better than the brute force method. Original Rijndael allows different key and block sizes of 128-bit, 192-bit, and 256-bits, but in the AES standard, only a 128-bit block size is allowed. However, key sizes of 128-bit, 192-bit, and 256-bit are allowed.

AES steps

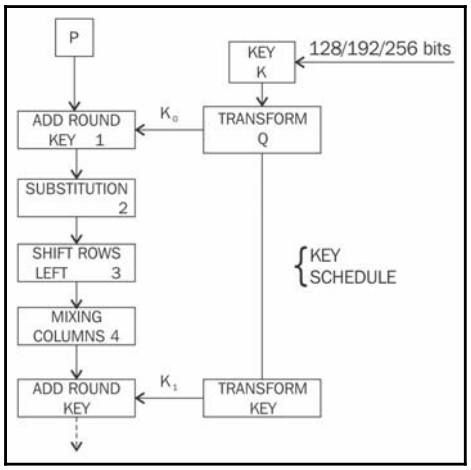
During the AES Algorithm processing, a 4 by 4 array of bytes knows as *state* is modified using multiple rounds. Full encryption requires 10 to 14 rounds depending on the size of the key. The following table shows the key sizes and the required number of rounds:

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 in order to encrypt the input. These stages are `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`:

1. In the `AddRoundKey` step, the state array is XORed with a subkey, which is derived from the master key.
2. This is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array.
3. This step is used to shift each row except the first one in the state array to the left in a cyclic and incremental manner.
4. Finally, all bytes are mixed in this 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 inverted back:



AES block diagram, showing 1st round, in last round 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.

An OpenSSL example of how to encrypt and decrypt using AES

```
:~/Crypt$ openssl enc -aes-256-cbc -in message.txt -out message.bin
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
:~/Crypt$ ls -ltr
total 12
```

```
-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
:~/Crypt$ cat message.bin

Salted__w    s_ў  h~[?              :~/Crypt$
 :~/Crypt$
```

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

```
:~/Crypt$ openssl enc -base64 -in message.bin -out message.b64
:~/Crypt$ 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
:~/Crypt$ cat message.b64
U2FsdGVkX193uByIcwZf0Z7J1at+4L+Fj8/uzeDAtJE=
:~/Crypt$
```

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

```
:~/Crypt$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec
enter aes-256-cbc decryption password:
:~/Crypt$ 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
:~/Crypt$ cat message.dec
datatoencrypt
:~/Crypt$
```

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

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

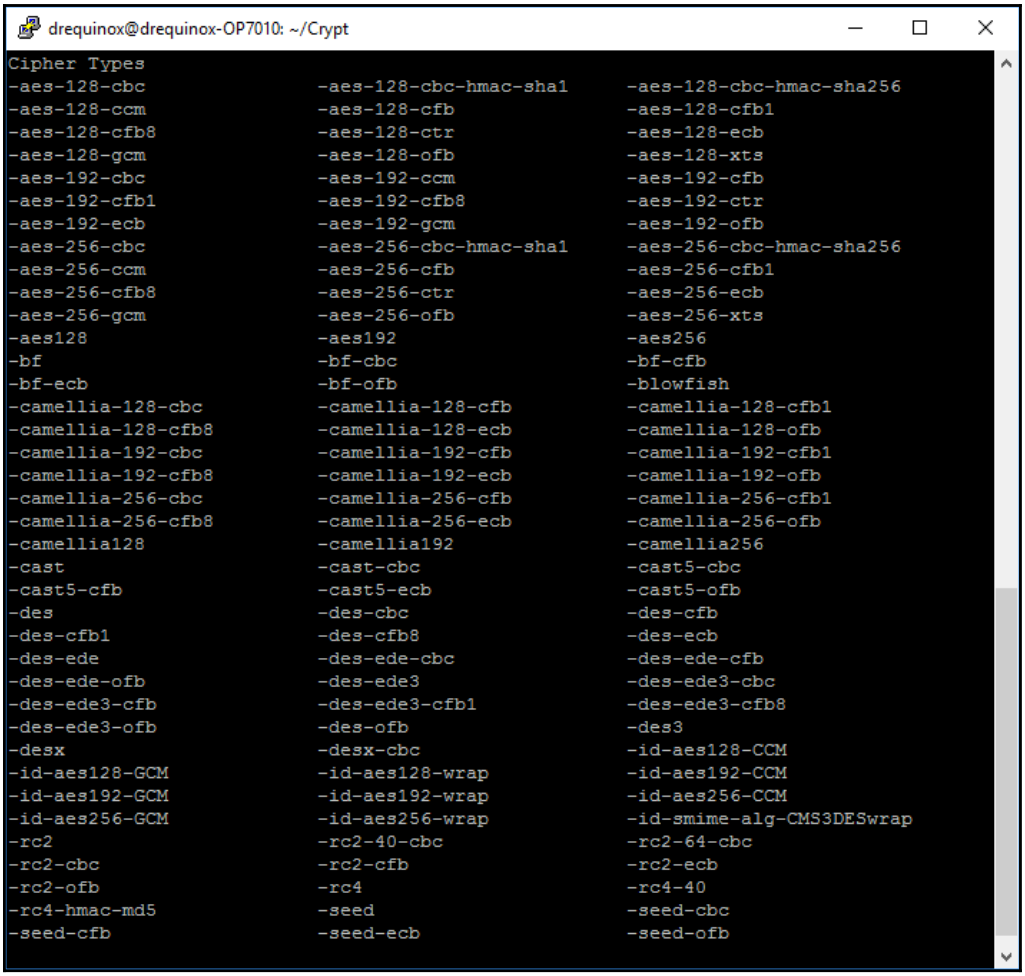
In order to decode from base64, the following commands are used. Take the `message.b64` file from the previous example:

```
:~/Crypt$ openssl enc -d -base64 -in message.b64 -out message.ptx
:~/Crypt$ 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

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

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

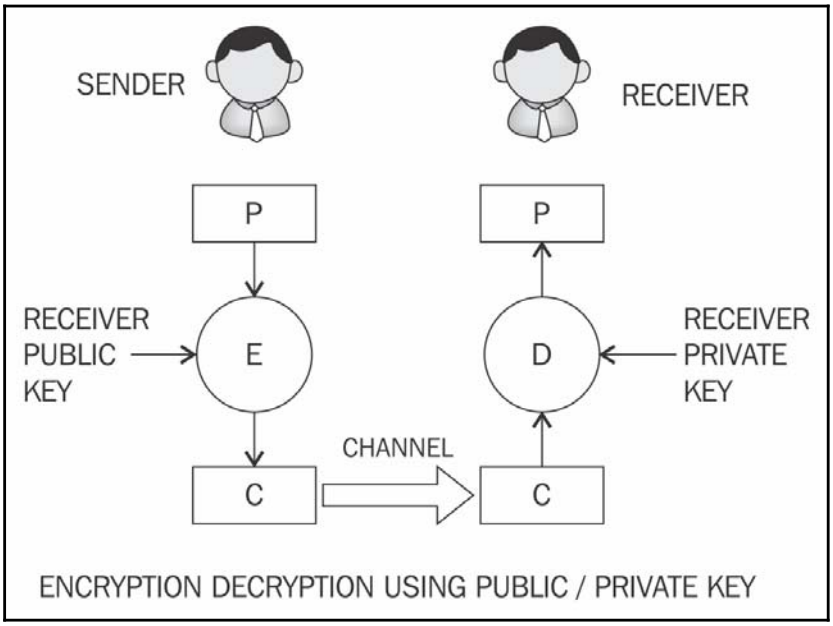


Screenshot displaying rich library options available in OpenSSL.

Asymmetric cryptography

Asymmetric cryptography refers to a type of cryptography whereby the key that is used to encrypt the data is different from the key that is used to decrypt the data. Also known as public key cryptography, it uses public and private keys in order to encrypt and decrypt data, respectively. Various asymmetric cryptography schemes are in use, such as RSA, DSA, and El-Gammal.

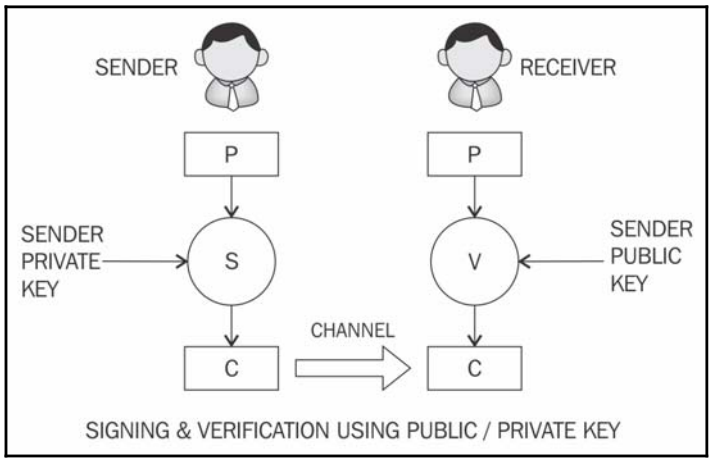
An overview of public key cryptography is shown in the following diagram:



Encryption decryption using public/private key

The diagram explains how a sender encrypts the data using a recipient's public key and is then transmitted over the network to the receiver. Once it reaches the receiver, it can be decrypted using the receiver's private key. 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.

Another diagram shows how public key cryptography can be used to verify the integrity of the received message by the receiver. 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 on the receiver's side, it can be verified for its integrity by the sender's public key. Note that there is no encryption being performed in this model. This model is only used for message authentication and validation purposes:



Model of a public key cryptography signature scheme

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

Key establishment mechanisms are concerned with the design of protocols that allow setting up of keys over an insecure channel. Non-repudiation service, a very desirable property in many scenarios, can be provided using digital signatures. Sometimes, it is important to not only authenticate a user, but to also 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 realized using public key cryptosystems, such as RSA, ECC, or El-Gammal.

Public key algorithms are slower in computation as compared to symmetric key algorithms. Therefore, they are not commonly used in the encryption of large files or the actual data that needs encryption. They are usually used to exchange keys for symmetric algorithms and 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 problems. There are three main families of asymmetric algorithms that are described here.

Integer factorization

These 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

This is based on a problem in modular arithmetic that it is easy to calculate the result of modulo function but it is computationally infeasible 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 finding 2, the exponent of the generator 3 is very hard. This hard problem is commonly used in **Diffie-Hellman** key exchange and digital signature algorithms.

Elliptic curves

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

$$y^2 = x^3 + ax + b$$

Here, *a*, *b* are integers that can have various values and are elements of the field on which the elliptic curve is defined. Elliptic curves can be defined over reals, rational numbers, complex numbers, or finite fields. For cryptographic purposes, 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*, *b*.

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

Public and private keys

In order to understand public key cryptography, the first concept that needs to be looked at is the idea of public and private keys.

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

A public key is the public part of the private-public key pair. A public key is available publicly 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 would be 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 regarding public keys, such as authenticity and identification of the publisher of the public keys.

RSA

A description of RSA is discussed here. RSA was invented in 1977 by *Ron Rivest*, *Adi Shamir*, and *Leonard Adelman*, hence the name RSA. This is based on the integer factorization problem, where the multiplication of two large prime numbers is easy but difficult to factor it back to the two original numbers.

The crux of the work in the RSA algorithm is during the key generation process. An RSA key pair is generated by performing the steps described here.

Modulus generation:

- Select p and q very large primes
- Multiply p and q , $n=p.q$ to generate modulus n

Generate co-prime:

- Assume a number called e.

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

Generate public key:

- Modulus generated in step 1 and e generated in step 2 is pair that, together, 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.

Generate private key:

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

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

Usually, an 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 calculate private key *d* easily, by applying the extended Euclidean algorithm, but someone who doesn't 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 very difficult (computationally infeasible) to factor.

Encryption and decryption using RSA

RSA uses the following equation to produce cipher text:

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

This means that plain text *P* is raised to *e* number of times and then reduced to modulo *n*.

Decryption in RSA is given by the following equation:

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

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 (ECC)

ECC is based on the discrete logarithm problem that is based on elliptic curves over finite fields (Galois fields). The main benefit of ECC over other types of public key algorithms is that it needs a smaller key size while providing the same level of security as, for example, RSA. Two notable schemes that originate from ECC are **Elliptic Curve Diffie-Hellman** (**ECDH**) for key exchange and **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) for digital signatures. It can also be used for encryption but is not usually used for this purpose in practice; instead, key exchange and digital signatures are more commonly used. As ECC needs less space to operate, it is becoming very popular on embedded platforms or in systems where storage resources are limited. As a comparison, the same level of security can be achieved in ECC by only using 256-bit operands as compared to 3072-bits in RSA.

Mathematics behind ECC

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

An elliptic curve can be defined as an equation here:

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

Here, *A* and *B* belong to a finite field *Zp* or *FP* (prime finite field) along with a special value called point of infinity. 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.

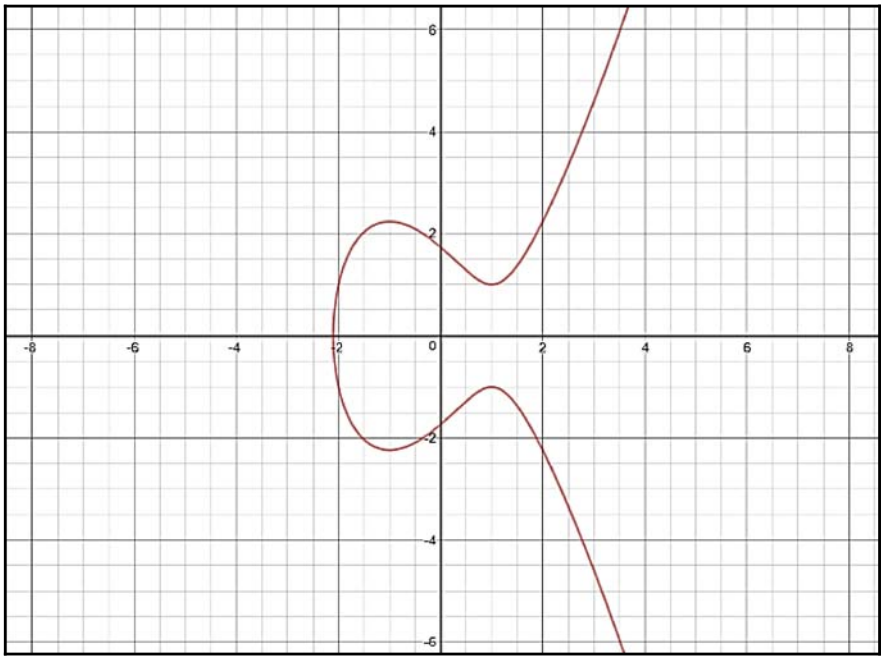
The condition is described here in the equation, which is a standard requirement that needs to be met. More precisely, this ensures that the curve is nonsingular:

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

A real number representation of elliptic curve can be visualized as shown in the following graph. This is a graph of equation over real numbers:

$$y^2 = x^3 + ax + b$$

The actual curves used in elliptic curve cryptography are over finite prime fields, but here, they are shown over real number as it becomes easier to visualize the operations when graphed over *R*:



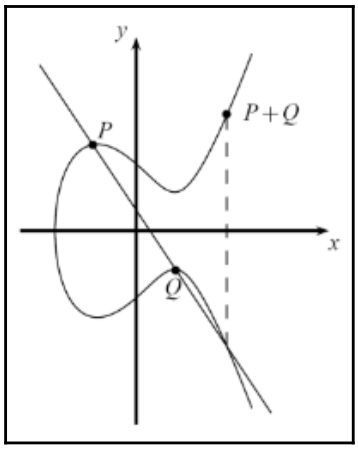
Elliptic curve over reals, a = -3 and b = 3

In order 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 earlier 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. Both of these operations can be visualized as shown in the following diagrams.

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 line is drawn through the curve that intersects the curve at two points shown below *P* and *Q*, which yields a third point between the curve and the line. This point is mirrored as *P+Q*, which represent the result of addition as *R*. This is shown as *P+Q* in the following diagram:



Point addition visualized over R

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

$$P + Q = R$$

In this case, two points are added in order 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:

$$(x1, y1) + (x2, y2) = (x3, y3)$$

The equation of point addition is as follows:

$$x_3 = s^2 - x_1 - x^2 \bmod p$$

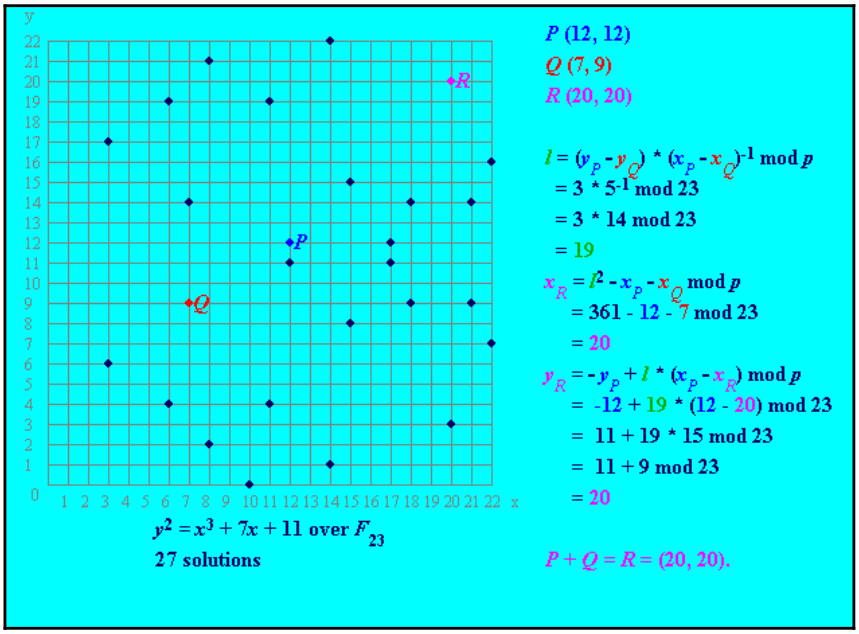
$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

Here, this is the result:

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

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

An example of point addition shown here is produced using Certicom's online calculator. This example shows the addition and solutions for the equation over finite field F_{23}. This is in contrast to the example shown earlier, which is over real numbers and only shows the curve but no solutions to the equation:



Example of point addition using Certicom's online calculator tool

In the example, the graph on the left-hand side shows the points that satisfy the equation shown here:

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

There are 27 solutions to the equation shown earlier over a finite field F_{23}. P and Q are chosen to be added to produce the point R. Calculations are shown on the right-hand 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 in the equation shows that the equation is satisfied indeed. This is shown as follows:

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

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

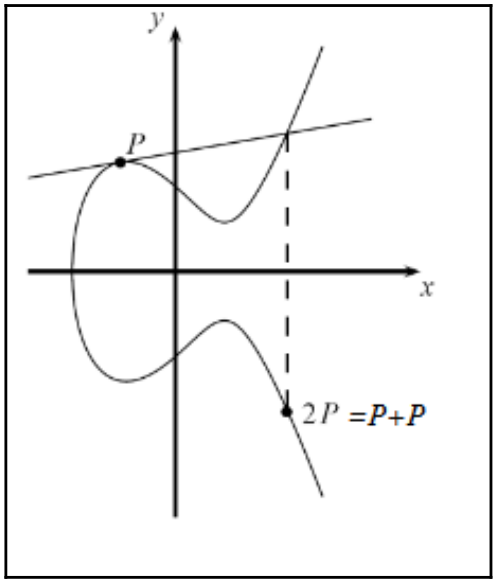
$$36\ mod\ 23 = 59\ mod\ 23$$

$$13 = 13$$

The next section will introduce 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 and is described in the following diagram. This is a process where P is added into 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 case of point doubling, the equation becomes as follows:

$$x_3 = s^2 - x_1 - x^2 \bmod p$$

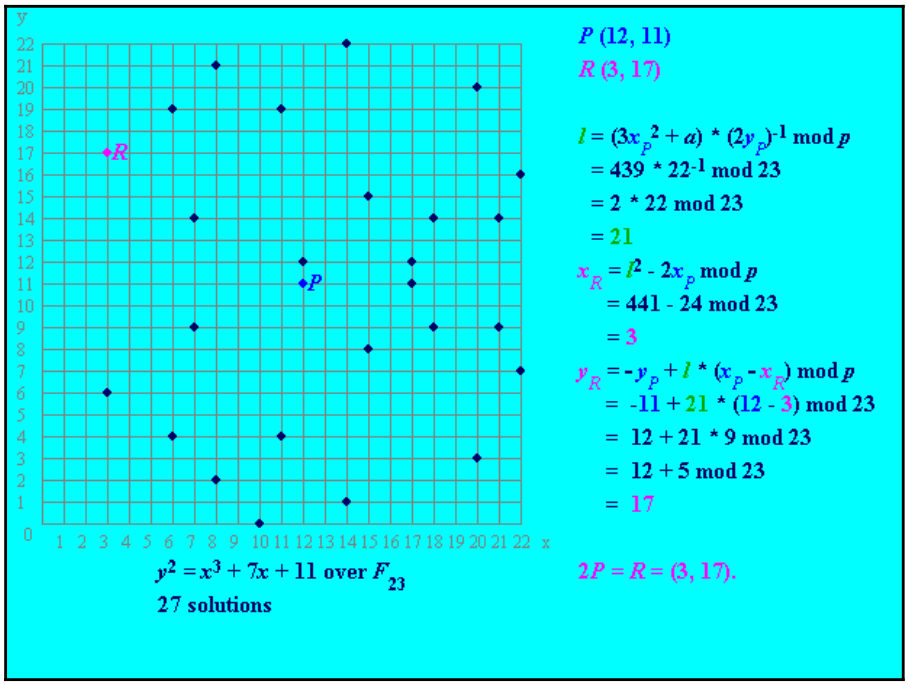
$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

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

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

An example is shown here, which shows the solutions and point doubling of elliptic curve over finite field F_{23}. The graph on the left-hand side shows the points that satisfy the equation:

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



Example of point doubling using certicom's online calculator tool

As shown earlier, on the right-hand side, a calculation is shown that finds the R after P is added into itself (point doubling). There is no Q as here, 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

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 chosen 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. An upcoming equation shows this 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:

$$P + P + \ldots + P = d\,P = T$$

Here, *T* is the public key (point on the curve) and d is the private key. In other words, public key is a random multiple of generator, whereas the private key is the integer that is used to generate the multiple. *#E* represents the order of the elliptic curve, which basically 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 at infinity.

A key pair is linked with specific domain parameters of an elliptic curve. Domain parameters include a 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 co-factor *h = #E(Fq)/n*. A practical example using OpenSSL will be described later in this section.

There are various parameters that are recommended and standardized to use as curves with ECC. You are shown an example of SECP256K1 specifications here. This is the specification that has been 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 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 FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

h = 01

Specification of SECP256K1 taken from http://www.secg.org/sec2-v2.pdf

An explanation of all these values in the sextuple is given here.

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 generator. Base point can be represented in either compressed or uncompressed form. There is no need to store all points on the curve in practical implementations. The compressed generator works because points on the curve can be identified by 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, an example using OpenSSL is shown to help you understand the practical aspects of RSA.

In the following section, it is shown how RSA public and private key pairs can be generated using OpenSSL.

How to generate public and private key pairs

First, it is shown how the RSA private key can be generated using OpenSSL.

Private key

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

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

```
~/Crypt$ 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-----
```

Generate public key

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

```
:~/Crypt$ openssl rsa -pubout -in privatekey.pem -out publickey.pem
writing RSA key
```

Public key can be viewed using a file reader or any text viewer, as shown here:

```
:~/Crypt$ 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 about the various components, such as modulus, prime numbers that are used in the process, exponents and coefficients of the generated private key, the following command can be used (the complete output is not shown as it is too large):

```
:~/Crypt$ 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
```

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

```
~/Crypt$ 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 us a message can use the public key to encrypt the message and send it to us. We can then use the corresponding private key to decrypt the file.

How to encrypt and decrypt using RSA with OpenSSL

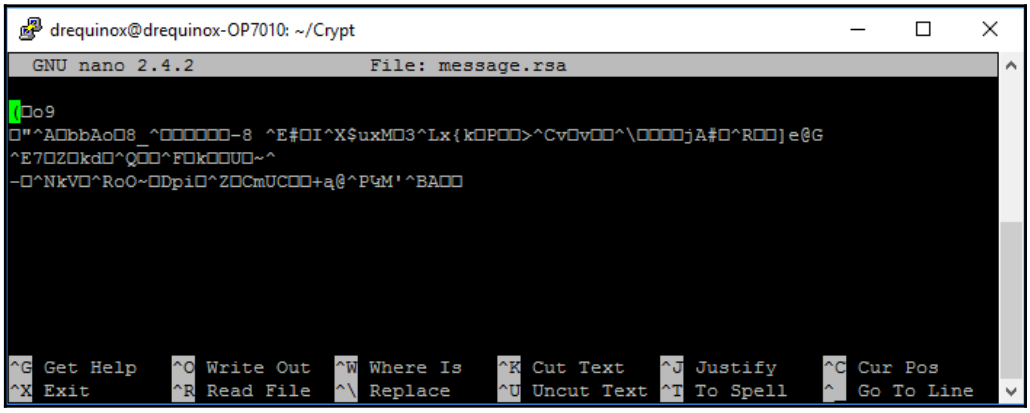
First in the section, an example will presented, which demonstrates how encryption can be performed using RSA.

Encryption

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

```
:~/Crypt$ 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 we open `message.rsa` in the nano editor, it will show some garbage:



message.rsa showing garbage data

Decrypt

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

```
:~/Crypt$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa -
out message.dec
```

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

```
:~/Crypt$ cat message.dec
datatoencrypt
```

ECC using OpenSSL

OpenSSL provides a very rich library of functions to perform elliptic curve cryptography. The following section shows how to practically use ECC functions in OpenSSL.

ECC private and public key pair

In this example, 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. We can see the list of all available standards' defined and recommended curves available in OpenSSL using the following command:

```
Crypt$ 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
```

As this produces a long output, the complete output is not shown and truncated in between. In the following example, SECP256k1 is used to demonstrate ECC usage.

Private key generation

```
~/Crypt$ openssl ecparam -name secp256k1 -genkey -noout -out ec-
privatekey.pem
~/Crypt$ 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 EC private key that is generated based on the SECP256K1 curve.

In order to generate a public key out of a private key, issue the following command:

```
~/Crypt$ 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:

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

Now the `ec-pubkey.pem` file contains the public key derived out of `ec-privatekey.pem`.

The private key can be further explored using the following command:

```
~/Crypt$ 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 explored further with the following command:

```
drequinox@drequinox-OP7010:~/Crypt$ 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
drequinox@drequinox-OP7010:~/Crypt$
```

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:

```
~/Crypt$ openssl ecparam -name secp256k1 -out secp256k1.pem
drequinox@drequinox-OP7010:~/Crypt$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
```

The file now contains all `SECP256K1` parameters and can be analyzed using the following command:

```
drequinox@drequinox-OP7010:~/Crypt$ openssl ecparam -in secp256k1.pem -text
-param_enc explicit -noout
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:
```

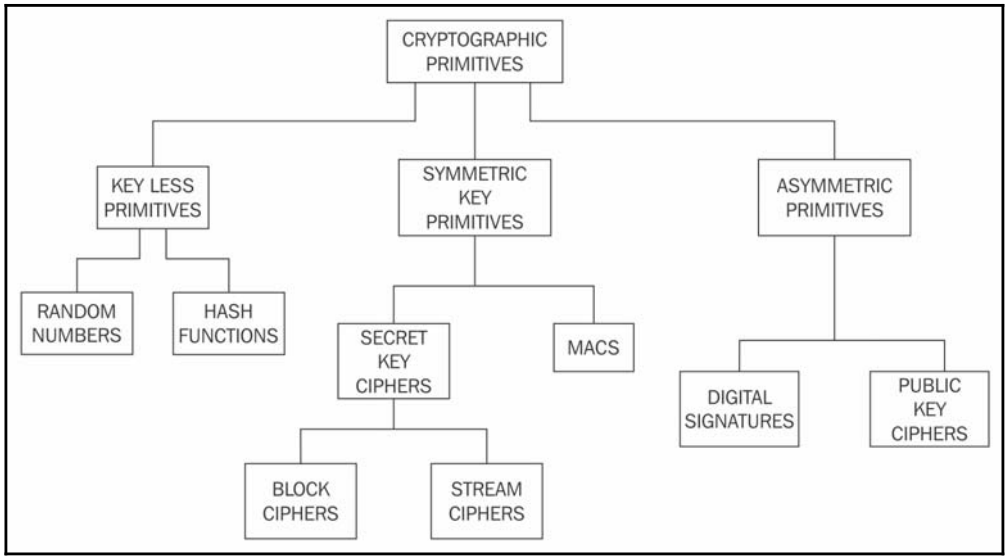
```
      36:41:41
Cofactor:  1 (0x1)
```

The preceding example shows the prime number used and values of `A` and `B` the with generator, order, and cofactor of the `SECP256K1` curve domain parameters.

There is another category of cryptographic primitives that is known as hash functions. Hash functions are not used to encrypt; data instead, they produce a fixed length digest of text.

Cryptographic primitives

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



Cryptographic primitives

Hash functions

Hash functions are used to create fixed length digests of arbitrarily long input strings. Hash functions are keyless and provide the data integrity service. They are usually built using iterated and dedicated hash function construction techniques. Various families of hash functions are available, such as MD, SHA1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used in digital signatures and message authentication codes, such as HMACs. They have three security properties, namely pre-image resistance, second pre-image resistance, and collision resistance. These properties are explained later in the section.

Hash functions are typically used to provide data integrity services. These can be used as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications used hash functions as a means of generating **pseudo random numbers** (**PRNGs**). Hash functions do not require a key. There are two practical and three security properties of hash functions that must be met depending on the level of requirements of integrity.

Compression of arbitrary messages into fixed length digest

This property is concerned with the fact that a hash function must be able to take a long input text of any length and output a fixed length compressed message. Hash functions produce a compressed output in various bit sizes, usually between 128-bits and 512-bits.

Easy to compute

Hash functions are efficient and fast one-way functions. The requirement is that they be very quick to compute regardless of the message size. The efficiency may decrease if the message is too big but the function should still be fast enough for practical use.

In the following section, security properties of hash functions are discussed.

Pre-image resistance

Consider an equation:

$$h(x) = y$$

Here, *h* is the hash function, *x* is the input, and *y* is the hash. The first security property requires that *y* cannot be reverse computed to *x*. *x* is considered a *pre-image* of *y*, hence the name pre-image resistance. This is also called one-way property.

Second pre-image resistance

This property requires that given *x* and *h(x)* , it is almost impossible to find any other message *m* , where *m != x* and *hash of m = hash of x*. *h(m) = h(x)*. This property is also known as weak collision resistance.

Collision resistance

This property requires that two different input messages should not hash to the same output. In other words, *h(x) != h(z)*. This property is also known as strong collision resistance.

Hash functions, due to their very nature, will always have some collisions, and that is where two different messages hash to the same output, but they should be computationally infeasible to find. A concept known as **avalanche effect** is desirable in all hash functions. Avalanche effect specifies that a small change, even a single character change in the input text, will result in a totally different hash output.

Hash functions are usually designed by following iterated hash functions approach. In this method, the input message is compressed in multiple rounds on a block-by-block basis to produce the compressed output. A popular type of iterated hash function is Merkle-Damgard construction. This construction is based on the idea of dividing the input data into equal sizes of blocks and then feeding them through the compression functions in an iterative manner. The collision resistance of the property of compression functions ensures that the hash output is also collision-resistant. Compression functions can be built using block ciphers. In addition to Merkle-Damgard, there are various other constructions of compression functions proposed by researchers, for example, *Miyaguchi-Preneel* and *Davies-Meyer*.

There are multiple hash function categories. You will be introduced to these categories in the upcoming section.

Message Digest (MD)

Message Digest functions were very popular in early 1990s. MD4 and MD5 are members of this category. Both MD functions are found to be insecure and not recommended for use any more. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms (SHAs)

SHA-0: This is a 160-bit function introduced by NIST in 1993.

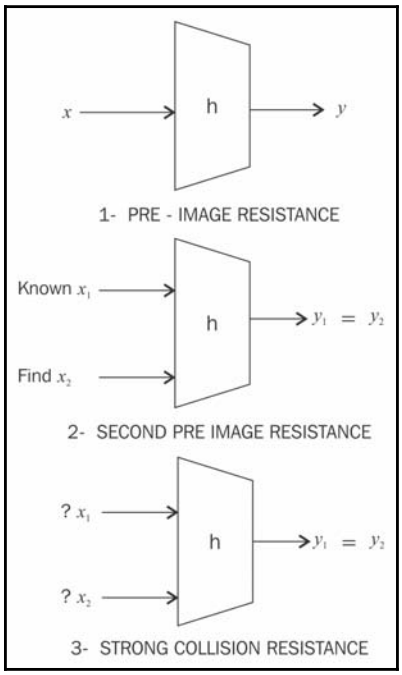
SHA-1: SHA-1 was introduced later by NIST as a replacement of SHA-0. This is also a 160-bit hash function. SHA-1 is used commonly in SSL and TLS implementations. It should be noted that SHA-1 is now considered insecure and is being deprecated by certificate authorities. Its usage is now discouraged in any new implementations.

SHA-2: This category includes four functions defined by the number of bits of the hash: SHA-224, SHA-256, SHA-384 and SHA-512.

SHA-3: This is the latest family of SHA functions. SHA3-224, SHA3-256, SHA3-384 and SHA3-512 are members of this family. SHA3 is a NIST-standardized version of Keccak. Keccak uses a new approach called *sponge construction* instead of the commonly used Merkle-Damgard transformation.

RIPEMD: RIPEMD is the acronym for *RACE Integrity Primitives Evaluation Message Digest*. It is based on the design ideas used to build MD4. There are multiple versions of RIPEMD, including 128-bit, 160-bit, 256-bit, and 320-bit.

Whirlpool: This is based on a modified version of Rijndael cipher known as W. It uses the Miyaguchi-Preneel compression function, which is a type of one-way function used for the compression of two fixed length inputs into a single fixed length output. It is a single block length compression function:



Three security properties of hash functions

Hash functions have many practical applications ranging from simple file integrity checks and password storage to be used in cryptographic protocols and algorithms. They are used in hash tables, distributed hash tables, bloom filters, virus finger printing, peer-to-peer P2P file sharing, and many other applications.

In blockchain, hash functions play a very vital role. Especially, the proof of work function uses SHA-256 twice in order to verify the computational effort spent by miners. RIPEMD 160 is used to produce bitcoin addresses. This will be discussed in more detail in later chapters.

Design of Secure Hash Algorithms (SHA)

In the following section, you will be introduced to the design of SHA-256 and SHA-3. Both of these are used in bitcoin and Ethereum, respectively. Ethereum doesn't use NIST Standard SHA-3 but Keccak, which is the original algorithm presented to NIST. NIST, after some modifications such as increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

SHA-256

SHA-256 has the input message size < 2^64-bits. Block size is 512-bits and has a word size of 32-bits. Output is 256-bit digest.

The compression function processes a 512-bit message block and a 256-bit intermediate hash value. There are two main components of this function: compression function and a message schedule.

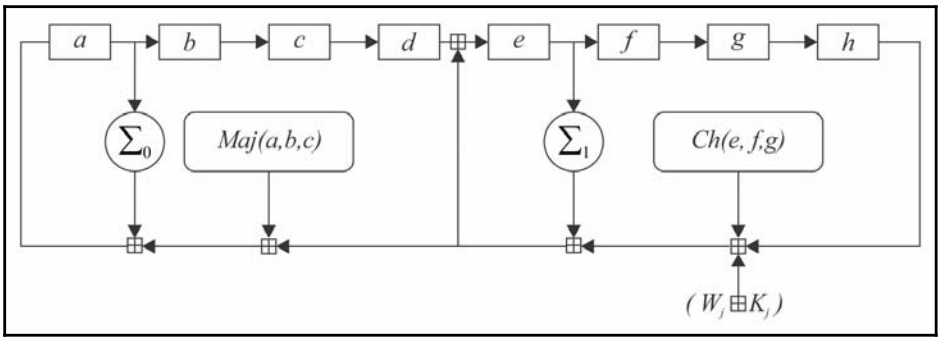
The algorithm works as follows:

- Pre-processing:

1. Padding of the message, which is used to make the length of a block to 512-bits if it is smaller than the required block size of 512-bits.
2. Parsing the message into message blocks that ensure that the message and its padding is divided into equal blocks of 512-bits.
3. Setting up the initial hash value, which is the eight 32-bit words obtained by taking the first 32-bits of the fractional parts of the square roots of the first eight prime numbers. These initial values are randomly chosen in order to initialize the process and gives a level of confidence that no backdoor exists in the algorithm.

- Hash computation:

1. Each message block is processed in a sequence and requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
2. First, the message schedule is prepared.
3. Then, eight working variables are initialized.
4. Then, the intermediate hash value is calculated.
5. Finally, the message is processed and the output hash is produced:



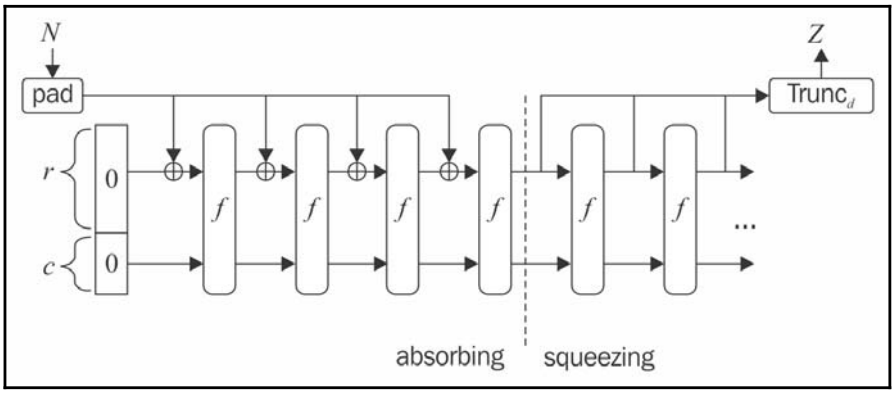
one round of SHA 256 compression function

In the preceding diagram, *a*, *b*, *c*, *d*, *e*, *f*, *g*, and *h* are the registers. **Maj** and **Ch** are applied bitwise. Σ_0 and Σ_1 performs bitwise rotation. Round constants are $\mathbf{W}_j$ and $\mathbf{K}_j$, which are added *mod 2^32*.

Design of SHA3 (Keccak)

The structure of SHA-3 is very different from the usual SHA-1 and SHA-2. The key idea behind SHA-3 is based on un-keyed permutations as opposed to other usual hash functions' constructions that used keyed permutations. Keccak also does not make use of the Merkle-Damgard transformation that is commonly used to handle arbitrary length input messages in hash functions. A newer approach called sponge and squeeze construction is used in Keccak, which is basically a random permutation model. Different variants of SHA3 have been standardized, such as SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. SHAKE128 and SHAKE256 are extendable output functions that are also standardized by `NIST.XOF` functions that allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model that is the basis of SHA3 or Keccak. As an analogy to sponge, first, the data is absorbed into the sponge after applying padding, where it is then changed into a subset of permutation state using XOR and then the output is squeezed out of the sponge function that represents the transformed state. Rate is the input block size of a sponge function, whereas capacity determines the generic security level:



SHA-3 absorbing and squeezing function in SHA3

OpenSSL example of hash functions

The following command will produce a hash of 256-bits of Hello messages using the SHA256 algorithm:

```
:~/Crypt$ echo -n 'Hello' | openssl dgst -sha256
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

Note that even a small change in the text, such as changing the case of *H*, results in a big change in the output hash. This is known as *avalanche effect*, as discussed earlier:

```
:~/Crypt$ echo -n 'hello' | openssl dgst -sha256
(stdin)= 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Note that both outputs are completely different:

```
Hello:
18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:80:07:
d1:76:48:26:38:19:69
hello:
2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5e:73:
04:33:62:93:8b:98:24
```

Message Authentication codes (MACs)

MACs are sometimes called keyed hash functions and can be used to provide message integrity and authentication. In others words, they are used to provide data origin authentication. These are symmetric cryptographic primitives using a shared key between the sender and the receiver. MACs can be constructed using block ciphers or hash functions.

MACs using block ciphers

In this approach, block ciphers are used in the **Cipher block chaining mode** (**CBC mode**) in order to generate a MAC. Any block cipher-for example, AES in the CBC mode-can be used. The MAC of the message is in fact the output of the last round of the CBC operation. The length of the MAC output is the same as the block length of the block cipher used to generate MAC. MACs are verified simply by computing the MAC of the message and comparing it with the received MAC. If they are the same, then the message integrity is confirmed; otherwise, the message is considered altered. It should also be noted that MACs work like digital signatures, but they cannot provide the nonrepudiation service due to their symmetric nature.

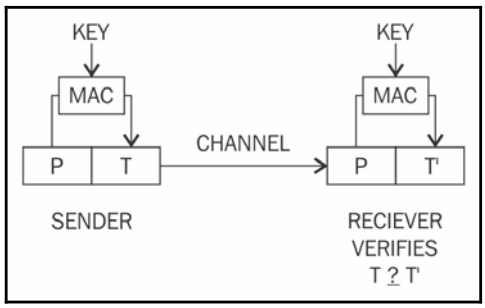
HMACs (hash-based MACs)

Similar to the hash function, they produce a fixed length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using MAC and the receiver verifies it using the shared key. The key is hashed with the message using either of the two methods known as secret prefix or the secret suffix method. In the first method, the key is concatenated with the message, that is, the key comes first and the message comes after, whereas in the latter method, the key comes after the message:

$$\textit{Secret prefix: } M = MACk(x) = h(k||x)$$

$$\textit{Secret suffix: } M=MACk(x) = h(x||k)$$

There are pros and cons of both methods. Some attacks on both schemes have been discovered. There are HMAC constructions schemes that use various techniques, such as **ipad** and **opad** (inner padding and outer padding) proposed by researchers that are considered secure with some assumptions:

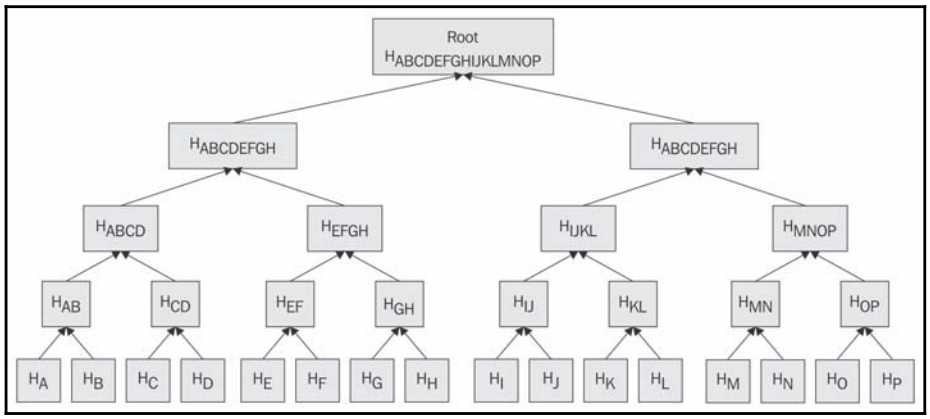


Operation of a MAC function

Merkle trees

The concept of Merkle tree was introduced by *Ralph Merkle*. A visualization of Merkle tree is shown here, which makes it easy to understand. Merkle trees allow secure and efficient verification of large data sets.

It is a binary tree in which first, the inputs are placed at the leaves (node with no children), and then values of pairs of child nodes are hashed together in order to produce a value for the parent node (internal node) until a single hash value known as Merkle root is achieved:



A Merkle tree

Patricia trees

In order to understand Patricia trees, first, you will be introduced to the concept of a **trie**. A trie or a digital tree is an ordered tree data structure used to store a dataset.

Practical Algorithm to Retrieve Information Coded in Alphanumeric (**Patricia**), also known as Radix tree, is a compact representation of a trie in which a node that is the only child of a parent is merged with its parent.

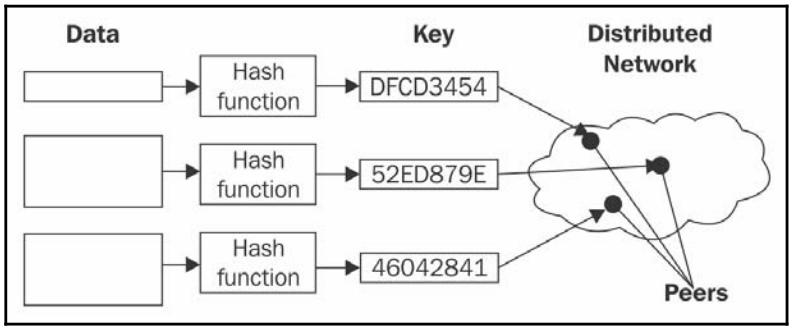
Merkle-Patricia tree, based on the definitions of Patricia and Merkle, is a tree that has a root node that contains the hash value of the entire data structure.

Distributed hash tables (DHTs)

A hash table is a data structure that is used to map keys to values. Internally, a hash function is used to calculate an index into an array of buckets, from which the required value can be found. Buckets have records stored in them using a hash key and are organized in a particular order.

With the definition provided earlier in mind, one can think of the distributed hash table as a data structure where data is spread across various nodes and nodes are equivalent to buckets in a peer-to-peer to network.

The following diagram visually shows how a DHT works. The example shows that data is passed through a hash function, which results in generating a compact key. This key is then linked with the data (values) on the peer-to-peer network. When users on the network request the data (via the filename), the filename can be hashed again to produce the same key and any node on the network can then be requested to find the corresponding data. DHTs provides decentralization, fault tolerance, and scalability:



Distributed hash tables

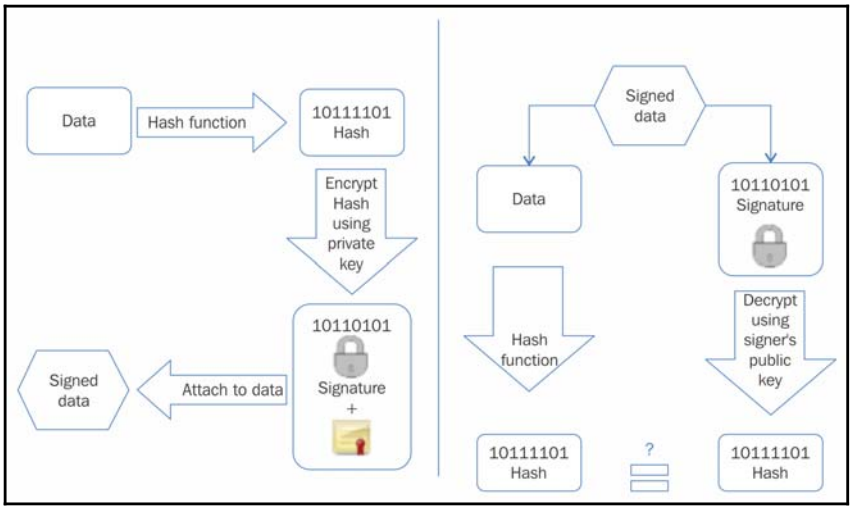
Digital signatures

Digital signatures provide a means of associating a message with an entity from which the message has been originated. Digital signatures are used to provide data origin authentication and nonrepudiation. They are calculated in two steps. High-level steps of an RSA digital signature scheme is given as follows:

1. Calculate the hash value of the data packet. This will provide the data integrity guarantee as hash can be computed at the receiver's end again and matched with the original hash to check whether the data has been modified in transit. Technically, message signing can work without hashing the data first, but is not considered secure.
2. The second step signs the hash value with the signer's private key. As only the singer has the private key, the authenticity of the signature and the signed data is ensured.

Digital signatures have some important properties, such as authenticity, unforgeability, and nonreusability. Authenticity means that the digital signatures are verifiable by a receiving party. The unforgeability property ensures that only the sender of the message is able to use the signing functionality using the private key. In other words, no one else should be able to produce the signed message that has been produced by the legitimate sender. Non reusability means that the digital signature cannot be separated from a message and used for another message again.

The operation of a generic digital signature function is shown in the following diagram:



Digital signing (left) and verification process (right) (Example of RSA digital signatures)

If a sender wants to send an authenticated message to a receiver, there are two methods that can be used. These two approaches to use digital signatures with encryption are introduced here.

Sign then encrypt

In this approach, the sender digitally signs the data using the private key, appends the signature to the data, and then encrypts the data and the digital signature using the receiver's public key. This is considered a more secure scheme as compared to the encrypt then sign scheme described next.

Encrypt then sign

In this approach, the sender encrypts the data using the receiver's public key and then digitally signs the encrypted data.

In practice, a digital certificate that contains the digital signature is issued by a **certificate authority** (**CA**) that associates a public key with an identity.

Various schemes, such as RSA, Digital Signature Algorithm, and Elliptic Curve Digital Signature Algorithm-based digital signature schemes are used in practice. RSA is the most commonly used; however, with the traction of elliptic curve cryptography, ECDSA-based schemes are also becoming quite popular.

The ECDSA scheme is described in detail here.

Elliptic Curve Digital signature algorithm (ECDSA)

In order to sign and verify using the ECDSA scheme, the first key pair needs to be generated:

1. First, define an elliptic curve E:
    1. With modulus P.
    2. Coefficients a and b.
    3. Generator point A that forms a cyclic group of prime order q.

2. An integer d is chosen randomly so that $0 < d < q$.
3. Calculate public key B so that $B = d\ A$.

Public key is the sextuple of the form shown here:

$$Kpb = (p,a,b,q,A,B)$$

Private key is randomly chosen d in Step 2:

$$Kpr = d$$

Now the signature can be generated using the private and public key.

1. First, an ephemeral key Ke is chosen, where $0 < Ke < q$. It should be ensured that Ke is truly random, and no two signatures have the same key; otherwise, the private key can be calculated.
2. Another value R is calculated using $R = Ke\ A$, that is, by multiplying A (the generator point) and the random ephemeral key.
3. Initialize a variable r with the x coordinate value of point R. $r = xR$.
4. The signature can be calculated as follows:

$$S = (h(m) + dr)K_{e^{-1}} \bmod q$$

Here, m is the message for which the signature is being computed and $h(m)$ is the hash of the message m.

Signature verification is carried out by following this process.

1. Auxiliary value w is calculated as $w = s^{-1} mod\ q$.
2. Auxiliary value $u1 = w.\ h(m)\ mod\ q$.
3. Auxiliary value $u2 = w.\ r\ mod\ q$.
4. Calculate Point P, $P = u1A + u2B$.
5. Verification is carried out as follows.
6. r, s is accepted as a valid signature if x-coordinate of the point P calculated in Step 4 has the same value as the signature parameter $r\ mod\ q$.

that is:

$$Xp = r \bmod q \text{ means valid signature}$$

Xp != r mod q means invalid signature

Various practical examples are shown here, which shows how the RSA digital signature can be generated, used, and verified using OpenSSL.

How to generate a digital signature

The first step is to generate a hash of the message file:

```
:~/Crypt$ openssl dgst -sha256 message.txt
SHA256(message.txt)=
eb96d1f89812bf4967d9fb4ead128c3b787272b7be21dd2529278db1128d559c
```

Both hash generation and signing can be done in a single step, as shown here. Note that `privatekey.pem` is generated in the steps provided previously:

```
:~/Crypt$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin
message.txt
```

Now let's display the directory showing relevant files:

```
:~/Crypt$ ls -ltr
total 36
-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  32 Sep 21 06:16 message.ptx
-rw-rw-r-- 1 drequinox drequinox 916 Sep 21 06:28 privatekey.pem
-rw-rw-r-- 1 drequinox drequinox 272 Sep 21 06:30 publickey.pem
-rw-rw-r-- 1 drequinox drequinox 128 Sep 21 06:43 message.rsa
-rw-rw-r-- 1 drequinox drequinox  14 Sep 21 06:49 message.dec
-rw-rw-r-- 1 drequinox drequinox 128 Sep 21 07:05 signature.bin
:~/Crypt$ cat signature.bin
V  _[  h] h t   + T~Ol  s (  Cϙ"# A Q U, uf   p* □ *7   T'  u eAy
$ x  <$ a  `  :L qWh  uG    = $  :~/Crypt$
```

In order to verify the signature, the following operation can be performed:

```
:~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature
signature.bin message.txt
Verified OK
:~/Crypt$
```

Similarly, if some other signature file that is not valid is used, the verification will fail, as shown here:

```
:~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature
someothersignature.bin message.txt
Verification Failure
```

Now you are introduced to an example that shows how OpenSSL can be used to perform ECDSA-related operations.

ECDSA using OpenSSL

First, the private key is generated using the following commands:

```
~/Crypt$ openssl ecparam -genkey -name secp256k1 -noout -out
eccprivatekey.pem
~/Crypt$ cat eccprivatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIMVmyrnEDOs7SYxS/AbXoIwqZqJ+gND9Z2/nQyzcpaPBoAcGBSuBBAAK
oUQDQgAEEKKS4E4+TATIeBX8o2J6PxKkjcoWrXPwNRo/k4Y/CZA4pXvlyTgH5LYm
QbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END EC PRIVATE KEY-----
```

Now the public key is generated out of the private key:

```
~/Crypt$ openssl ec -in eccprivatekey.pem -pubout -out eccpublickey.pem
read EC key
writing EC key
~/Crypt$ cat eccpublickey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEEKKS4E4+TATIeBX8o2J6PxKkjcoWrXPw
NRo/k4Y/CZA4pXvlyTgH5LYmQbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END PUBLIC KEY-----
~/Crypt$
```

Now suppose a file named `testsign.txt` needs to be signed and verified. This can be achieved as follows:

1. Create a test file:

```
~/Crypt$ echo testing > testsign.txt
~/Crypt$ cat testsign.txt
testing
```

2. Run the following command to generate a signature using a private key for the `testsign.txt` file:

```
~/Crypt$ openssl dgst -ecdsa-with-SHA1 -sign eccprivatekey.pem
testsign.txt > ecsign.bin
```

3. Finally, the command for verification can be run as shown here:

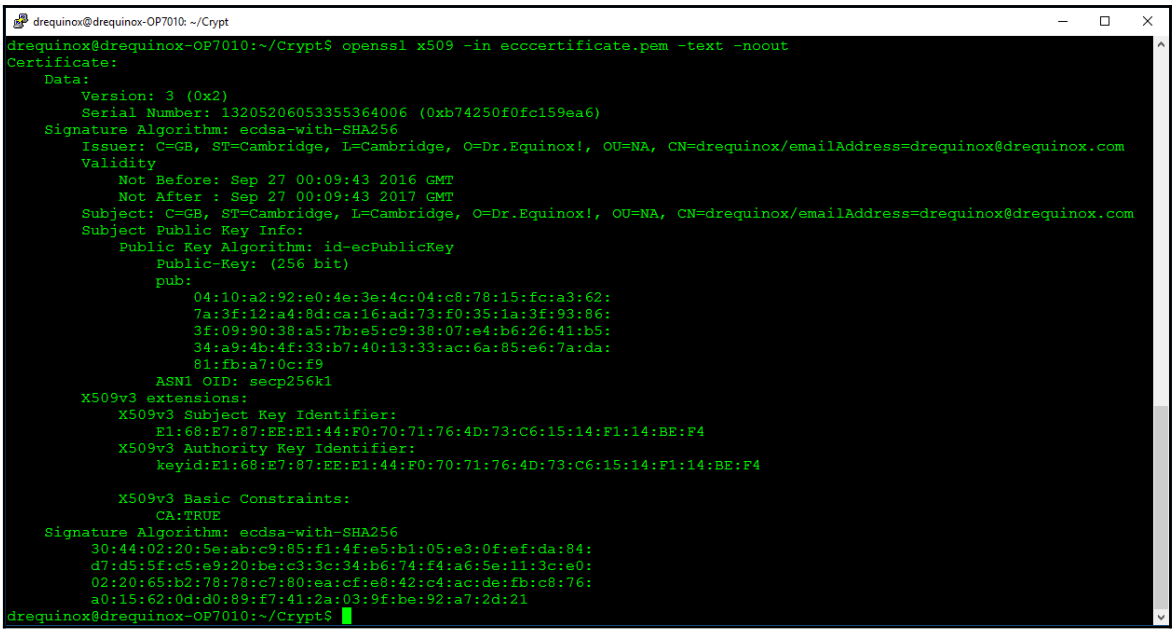
```
~/Crypt$ openssl dgst -ecdsa-with-SHA1 -verify eccpublickey.pem
-signature ecsign.bin testsign.txt
Verified OK
```

A certificate can also be generated using the private key generated earlier:

```
~/Crypt$ openssl req -new -key eccprivatekey.pem -x509 -nodes -days 365 -
out ecccertificate.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:Cambridge
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dr.Equinox!
Organizational Unit Name (eg, section) []:NA
Common Name (e.g. server FQDN or YOUR name) []:drequinox
Email Address []:drequinox@drequinox.com
```

The certificate can be explored using the command below:

```
~/Crypt$ openssl x509 -in ecccertificate.pem -text -noout
```



X509 certificate that uses ECDSA algorithm with SHA-256

There are other topics in cryptography that are presented here due to their relevance to blockchain or potential use in future blockchain ecosystems.

Homomorphic encryption

Usually, public key cryptosystems, such as RSA, are multiplicative homomorphic or additive homomorphic, such as Paillier cryptosystem, and are called **partially homomorphic** systems. Additive PHEs are suitable for e-voting and banking applications. Until recently, there has been no system that supported both operations, but in 2009, a **fully homomorphic** system was discovered by *Craig Gentry*. As these schemes allow the processing of encrypted data without the need for decryption, they have many different possible applications, especially in scenarios where privacy is required to be maintained but data is also required to be processed by potentially untrusted parties, for example, cloud computing and online search engines. Recent development in homomorphic encryption has been very promising and researchers are actively working to make it efficient and more practical. This is of particular interest in the blockchain technology, as described later in the book, because it can solve the problem of confidentiality and privacy in blockchain.

Signcryption

Signcryption is a public key cryptography primitive that provides all the functions of the digital signature and encryption. It was invented by *Yuliang Zheng* and is now an ISO standard ISO/IEC 29150:2011. Traditionally, signature then encrypt or encrypt then sign schemes are used to provide unforgeability, authentication, and nonrepudiation, but with Signcryption, all services of digital signatures and encryption are provided with cost less than that of sign then encrypt schemes.

This is **Cost (signature & encryption) << Cost (signature) + Cost (Encryption)** in a single logical step.

Zero knowledge proofs

Zero knowledge proofs were introduced by *GoldWasser*, *Micali*, and *Rackoff*. These proofs are used to prove the validity of an assertion without revealing any information whatsoever about the assertion. There are three properties of ZKPs that are required, namely completeness, soundness, and zero-knowledge property.

Completeness ensures that if a certain assertion is true, then the verifier will be convinced of this claim by the prover. The soundness property makes sure that if an assertion is false, then no dishonest prover can convince the verifier otherwise. Zero-knowledge property, as the name implies, is the key property of zero knowledge proofs whereby it is ensured that absolutely nothing is revealed about the assertion except whether it is true or false.

Zero knowledge proofs have sparked a special interest among researchers in the blockchain space due to its privacy properties that are very much desirable in financial and many other fields, such as law and medicine. A recent example of the successful implementation of the zero knowledge proof mechanism is the Zcash crypto currency. In Zcash, a specific type of zero knowledge proof, known as **zero-knowledge Succinct Non-interactive Argument of Knowledge** (**ZK-Snark**), is implemented. This will be discussed in detail in `Chapter 5`, *Alternative Coins*.

Blind signatures

Blind signatures were introduced by *David Chaum* in 1982 and are based on public key digital signature schemes, such as RSA. The key idea behind blind signatures is to get the message signed by the signer without actually revealing the message. This is achieved by disguising or blinding the message before signing it, hence the name blind signatures. This blind signature can then be verified against the original message just like a normal digital signature. Blind signatures were introduced as a mechanism to allow the development of digital cash schemes.

Encoding schemes

Other than cryptographic primitives, binary to text encoding schemes are also used in various scenarios. The most common usage is to convert binary data into text so that it can be either processed, saved, or transmitted via a protocol that does not support the processing of binary data. For example, sometimes, images are stored in the database as base64 encoding, which allows a text field to be able to store a picture. A commonly used encoding scheme is base64. Another encoding named base58 was popularized by its usage in bitcoin.

Cryptography is a vast field and this section has introduced basic concepts that are essential to understand cryptography in general and specifically from the blockchain and cryptocurrency point of view. In the next section, you are introduced to basic financial markets concepts.

The upcoming section describes general terminologies about trading, exchanges, and trade life cycle. More relevant information will be provided in later chapters where specific use cases are discussed.

Financial markets and trading

Financial markets exist to facilitate the transfers of savings from savers to investors. In an economic system, there are two sectors, namely household and business. Financial markets, at their core, act as an intermediary between the savers and the investors. Basically, there are three types of markets, namely money markets, credit markets, and capital markets. Money markets are short-term markets where money is lent to companies or banks to do interbank lending. Foreign exchange or FX is another category of money markets where currencies are traded. Credit markets consist mostly of retail banks where they borrow money from central banks and loan it to companies or households in the form of mortgages or loans.

Capital markets facilitate the buying and selling of financial instruments, mainly stocks and bonds. Capital markets can be divided into two types, primary and secondary markets. Stocks are issued directly by the companies to investors in primary markets, whereas in secondary markets, investors resell their securities to investors via stock exchanges. Various electronic trading systems are used by exchanges to facilitate the trading of financial instruments.

Trading

A market is a place where traders come to trade. It can either be a physical location or an electronic virtual location. Various financial instruments, including equities, stocks, foreign exchange, commodities, and various types of derivatives are traded at these marketplaces. Recently, many financial institutions have introduced software platforms to trade various types of instruments from different asset classes.

Trading can be defined as an activity in which traders buy or sell various financial instruments to generate profit and hedge risk. Investors, borrowers, hedgers, asset exchangers, and gamblers are a few types of traders. Traders have a short position when they owe something, for example, if they have sold a contract and have a long position when they buy a contract. There are various ways to transact trades, such as through brokers or directly on the exchange or over the counter. Brokers are agents who arrange trades for their customers. Brokers act on clients' behalf to deal at a given price or at the best possible price.

Exchanges

Exchanges are usually considered to be a very safe, regulated, and reliable place for trading. Recently, electronic trading has gained high popularity as compared to traditional floor-based trading. Now traders send orders to a central electronic order book from where the orders, prices, and related attributes are published to all associated systems using communication networks thus, in essence, creating a virtual marketplace. Exchange trades can be performed only by members of the exchange. In order to trade without these limitations, the counter parties can participate in **OTC** (**Over the Counter**) trading directly.

Orders and order properties

Orders are instructions to trade and are the main building blocks of a trading system. They have the following general attributes:

1. The instrument name.
2. Quantity.
3. Direction (buy or sell).

4. The type of the order that represents various conditions, for example, limit orders and stop orders, an example of which is 1500 Royal Bank of Scotland ordinary shares for GBP £15.50.

Orders are traded on the basis of bid prices and offer prices. Traders show their intention to buy or sell by attaching bid and offer prices to their orders. The price at which a trader will buy is known as the bid price. The price at which a trader is willing to sell is known as the offer price.

Order management and routing systems

Order routing systems routes and deliver orders to various destinations depending on the business logic. Customers use them to send orders to their brokers, who then send these orders to dealers, clearing houses, and exchanges.

There are different types of orders; the two most common ones are markets orders and limit orders. A market order is an instruction to trade at the best price currently available in the market, and these orders get filled immediately at spot prices. On the other hand, a limit order is an instruction to trade at the best price available but only if it is not lower than the limit price set by the trader. This can also be higher depending on the direction of the order, either sell or buy. All these orders are managed in an order book, which is a list of orders maintained by an exchange, and records the intention of buying or selling by the traders.

A position is a commitment to sell or buy an amount of financial instruments, such as securities, currencies, or commodities for a given price. The contracts, securities, commodities, and currencies that traders buy or sell are commonly known as trading instruments and come under the large umbrella of asset classes. The most common classes are real assets, financial assets, derivative contracts, and insurance contracts.

Components of a trade

A trade ticket is the combination of all details related to a trade. However, there is some variation depending on the type of the instrument and asset class, but generally, all instruments have the attributes discussed in the next section.

The underlying instrument
The underlying instrument is the basis of the trade. It can be a currency, a bond, interest rate, commodity, or equities.

General attributes

This includes general identification information and basic features associated with every trade. Common attributes include a unique ID, instrument name, type, status, trade date, and time.

Economic

These are features related to the value of the trade, for example, buy or sell value, ticker, exchange, price, and quantity.

Sales

Sales refers to the sales-characteristic-related details, such as the name of the sales person, and is just an information field, usually without any impact on the trade life cycle.

Counterparty

Counterparty is an important component of a trade as it shows the other side of the trade and is required to settle the trade successfully. Usual attributes include counterparty name, address, payment type, any reference IDs, settlement date, and delivery type.

Trade life cycle

A general trade life cycle includes various stages from order placement to execution and then settlement. This life cycle is described step by step as follows:

- **Pre-execution**: An order is placed at this stage.
- **Execution and booking**: When the order is matched and executed, it converts into a trade. At this stage, the contract between counter parties is matured.
- **Confirmation**: This is where both counter parties agree to particulars of the trade.
- **Post booking**: This stage is concerned with various scrutiny and verification processes to ascertain the correctness of the trade.
- **Settlement**: This is the most vital part during trade and at this stage, the trade is final.

- **Overnight (end of day processing)**: End of day processes include report generation, profit and loss calculations, and various risk calculations.

- In all the mentioned processes, many people and business functions are involved. Most commonly, these functions are divided into functions such as front office, middle office, and back office.

In the following section, you are introduced to some concepts that are essential in order to understand the strict and necessary rules and regulations that govern the financial industry. Some concepts are described here and in later chapters when specific use cases are discussed, and these ideas will help you understand the scenarios described.

Order anticipators

Order anticipators try to make profit before other traders can carry out trading. This is based on the anticipation where a trader knows how trading activities of other trades will affect prices. Frontrunners, sentiment-oriented technical traders, and squeezers are some examples of order anticipators.

Market manipulation

Market manipulation is strictly illegal in the UK and other countries. Fraudulent traders can spread false information in the market, which can result in price movements thus making illegal profits. Usually, manipulative market conduct is trade-based and it includes generalized and time-specific manipulations. Actions that can create an artificial shortage of stock, impression of false activity, and price manipulation to gain criminal benefit are included in this category.

Both of the terms discussed earlier are relevant to financial crime. and there is a possibility of developing blockchain-based systems that can thwart market abuse. This will be discussed in detail in later chapters, where specific use cases will be discussed.

Summary

This chapter aimed at introducing concepts of cryptography and financial markets in order to provide background information for you to be able to understand the material provided in later chapters. First, you were introduced to the basics of cryptography, and then various schemes such as symmetric and asymmetric ciphers were introduced. Practical examples using OpenSSL command line were shown so that you could experiment with various commands and experience various cryptographic functions firsthand. Also, some mathematical background was provided at the beginning of the chapter and where necessary, especially with the elliptic curve cryptography. All cryptography concepts presented in this chapter are related to the blockchain technology and are implemented or have been proposed to be implemented in various blockchains, cryptocurrencies, and relevant ecosystems. Moreover, you were given a quick introduction to the financial industry as it sets the scene for various examples that will be discussed in relation to the distributed ledger technology later in the book. As cryptography and finance are vast subjects, the material covered in his chapter is aimed to be introductory in nature (with some exceptions) and specific topics will be expanded upon in more detail, where relevant and required, in the next chapters.