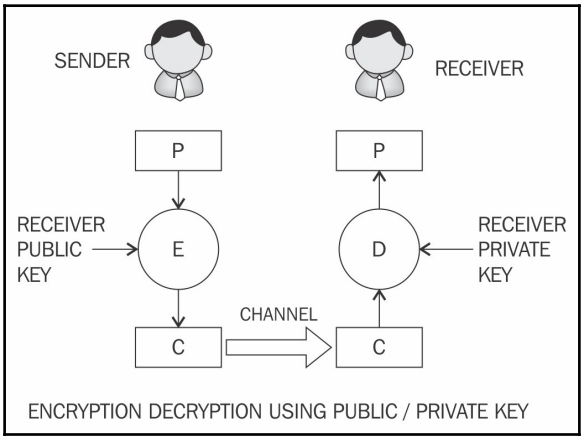
4
Public Key Cryptography

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

Asymmetric cryptography

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

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

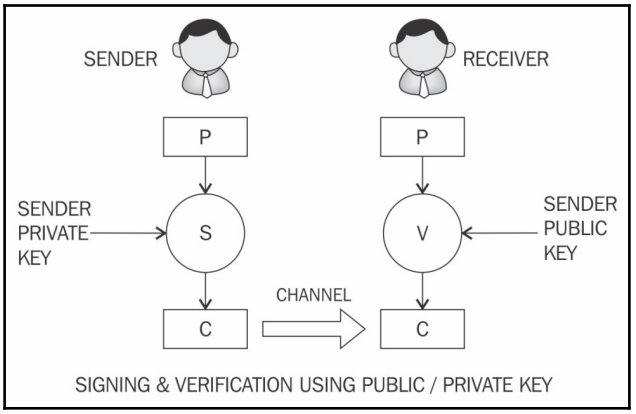


Encryption/decryption using public/private keys

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

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

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



Model of a public-key cryptography signature scheme

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

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

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

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

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

Integer factorization

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

Discrete logarithm

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

For example, consider the following equation:

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

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

Elliptic curves

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

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

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

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

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

Public and private keys

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

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

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

RSA

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

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

1. **Modulus generation**:

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

2. **Generate co-prime**:

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

3. **Generate the public key**:

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

4. **Generate the private key**:

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

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

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

Encryption and decryption using RSA

RSA uses the following equation to produce ciphertext:

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

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

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

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

Elliptic Curve Cryptography

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

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

Mathematics behind ECC

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

An elliptic curve is defined in the following equation:

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

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

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

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

$$4a^3 + 27b^2 \neq 0 \, mod \; p$$

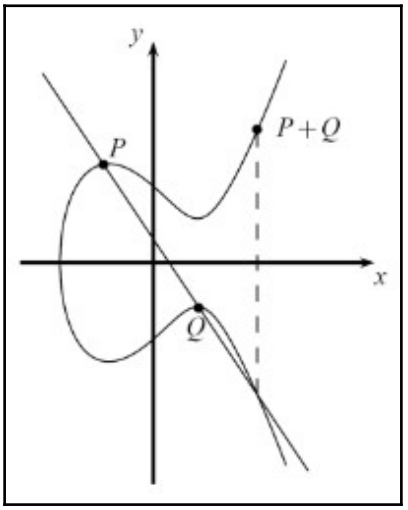
To construct the discrete logarithm problem based on elliptic curves, a large enough cyclic group is required. First, the group elements are identified as a set of points that satisfy the previous equation. After this, group operations need to be defined on these points.

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

Point addition

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

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



Point addition over R

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

$$P + Q = R$$

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

$$P + Q = R$$

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

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

The equation of point addition is as follows:

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

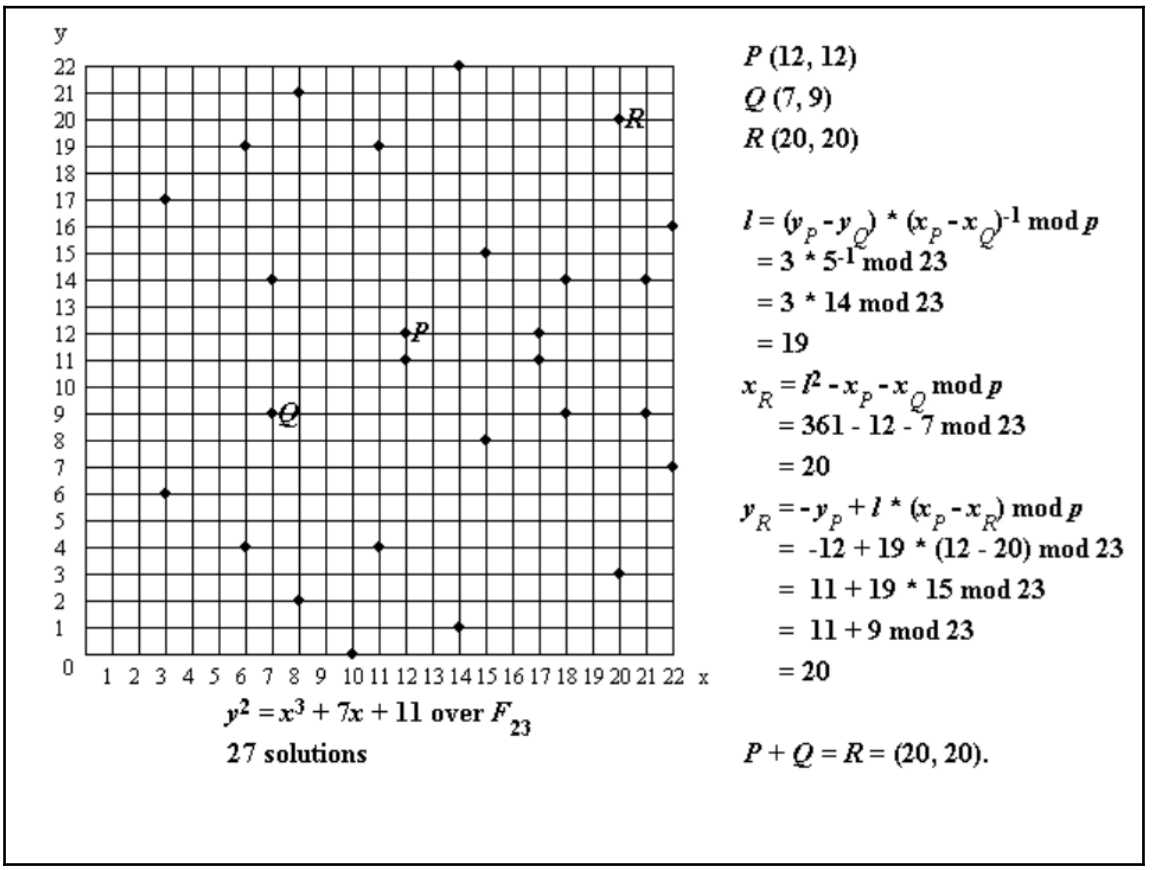
$$y_3 = s\,(x_1 - x_3) - y_1 \bmod p$$

Here, we see the result of the preceding equation:

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

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

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



Example of point addition

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

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

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

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

Using these values shows that the equation is indeed satisfied:

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

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

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

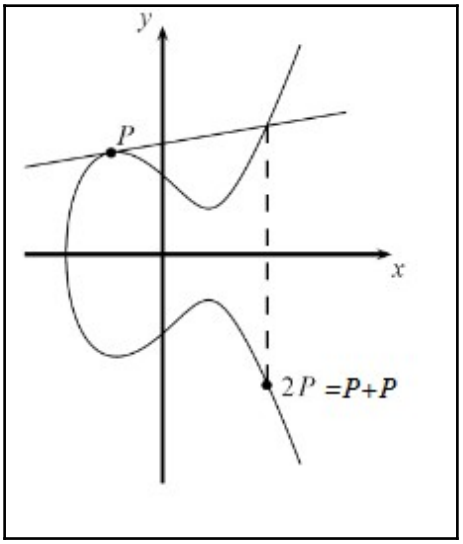
$$13 = 13$$

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

Point doubling

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

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



Graph representing point doubling over real numbers

In the case of point doubling, the equation becomes:

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

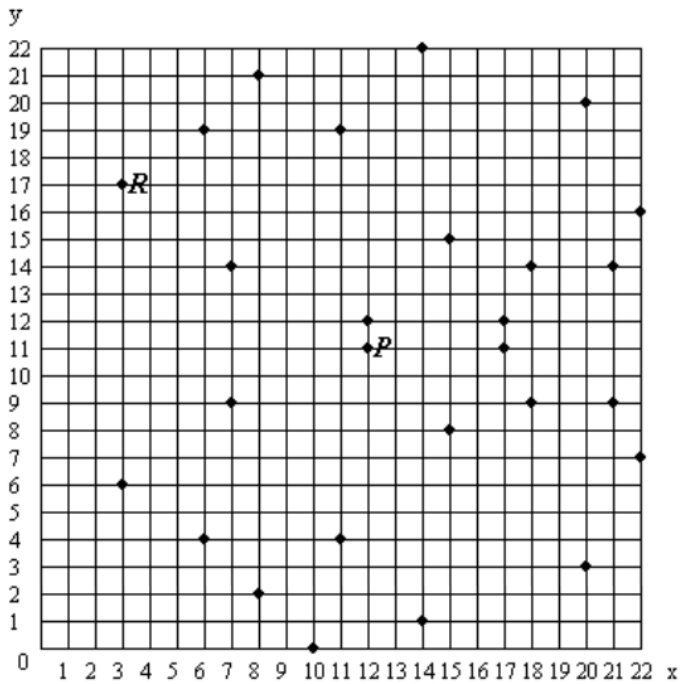
$$y_3 = s(x_1 - x_3) - y_1 \ mod \ p$$

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

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

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

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



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

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

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

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

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

Example of point doubling

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

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

Discrete logarithm problem in ECC

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

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

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

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

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

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

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

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

p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F

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

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

a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

The base point G in compressed form is:

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

and in uncompressed form is:

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

Finally the order n of G and the cofactor are:

n = FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

h = 01

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

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

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

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

RSA using OpenSSL

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

RSA public and private key pair

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

Private key

Execute the following command to generate the private key:

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

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

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

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

Public key

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

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

writing RSA key
```

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

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

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

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

Exploring the public key

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

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

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

Encryption and decryption

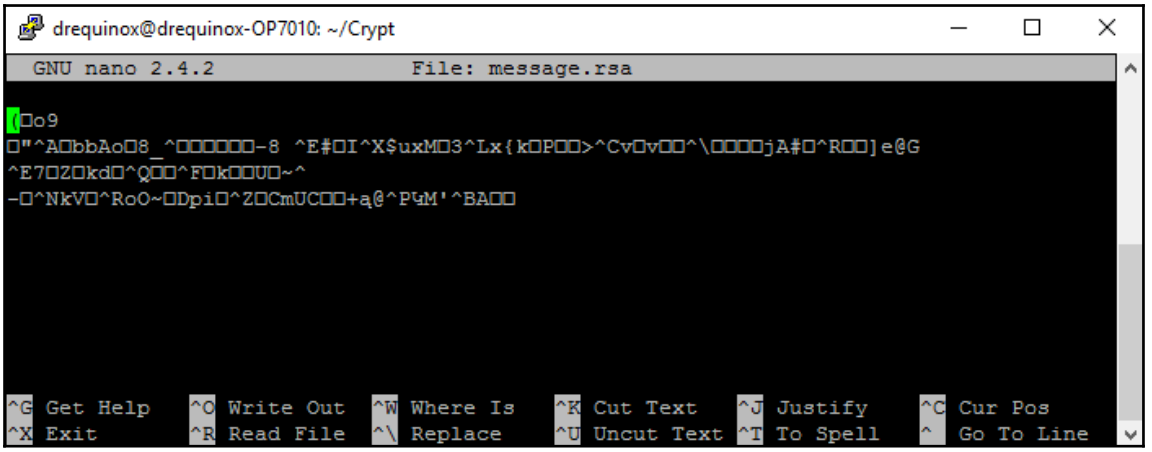
In this section, an example will be presented that demonstrates how encryption and decryption operations can be performed using RSA with OpenSSL.

Encryption

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

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

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



message.rsa showing garbage (encrypted) data

Decryption

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

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

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

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

ECC using OpenSSL

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

ECC private and public key pair

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

Private key

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

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

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

Private key generation

To generate the private key, execute the following command:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Hash functions

Hash functions are used to create fixed-length digests of arbitrarily-long input strings. Hash functions are keyless, and they 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, SHA-1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used for digital signatures and **Message Authentication Codes** (**MACs**), such as HMACs. They have three security properties, namely preimage resistance, second preimage resistance, and collision resistance. These properties are explained later in this section.

Hash functions are also typically used to provide data integrity services. These can be used both as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications use hash functions as a means for generating **Pseudo-random Numbers Generator** (**PRNGs**). There are two practical and three security properties of hash functions that must be met depending on the level of integrity required. These properties are discussed in the following subsections.

Compression of arbitrary messages into fixed-length digest

This property relates to the fact that a hash function must be able to take an 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. It is required that hash functions 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.

Preimage resistance

This property can be explained by using the simple equation shown as follows:

$$h(x) = y$$

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 preimage of *y*, hence the name **preimage resistance**. This is also called a one-way property.

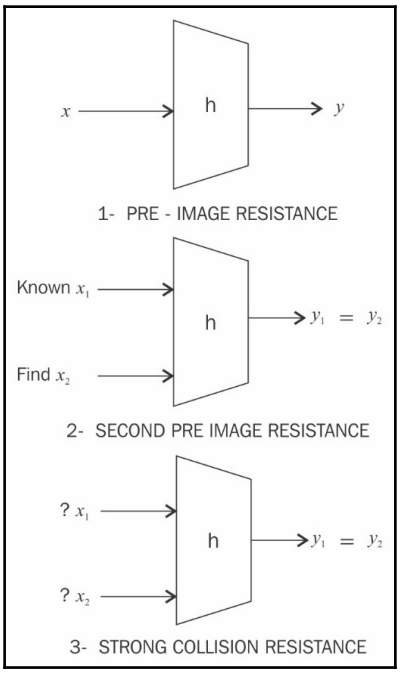
Second preimage resistance

The **second preimage resistance** 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* or *h(m) = h(x)*. This property is also known as **weak collision resistance**.

Collision resistance

The **collision resistance** 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**.

All these properties are shown in the following diagram:



Three security properties of hash functions

Due to their very nature, hash functions will always have some collisions. This is where two different messages hash to the same output. However, they should be computationally impractical to find. A concept known as the **avalanche effect** is desirable in all hash functions. The avalanche effect specifies that a small change, even a single character change in the input text, will result in an entirely different hash output.

Hash functions are usually designed by following an iterated hash functions approach. With this method, the input message is compressed in multiple rounds on a block-by-block basis in order 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 block sizes 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.

Multiple categories of hash function are introduced in the following subsections.

Message Digest

Message Digest (**MD**) functions were prevalent in the early 1990s. MD4 and MD5 fall into this category. Both MD functions were found to be insecure and are not recommended for use anymore. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms

The following list describes the most common **Secure Hash Algorithms** (**SHAs**):

- **SHA-0**: This is a 160-bit function introduced by NIST in 1993.
- **SHA-1**: SHA-1 was introduced in 1995 by NIST as a replacement for 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 it is being deprecated by certificate authorities. Its usage is 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. SHA-3-224, SHA-3-256, SHA-3-384, and SHA-3-512 are members of this family. SHA-3 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 the 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.

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

Hash functions play a vital role in blockchain. Especially, The PoW function in particular 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 further in later chapters.

In the next section, the design of the SHA algorithm is introduced.

Design of Secure Hash Algorithms

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 does not use NIST Standard SHA-3, but Keccak, which is the original algorithm presented to NIST. NIST, after some modifications, such as an increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

Design of SHA-256

SHA-256 has the input message size < 2^{64}-bits. Block size is 512-bits, and it has a word size of 32-bits. The output is a 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: the compression function and a message schedule.

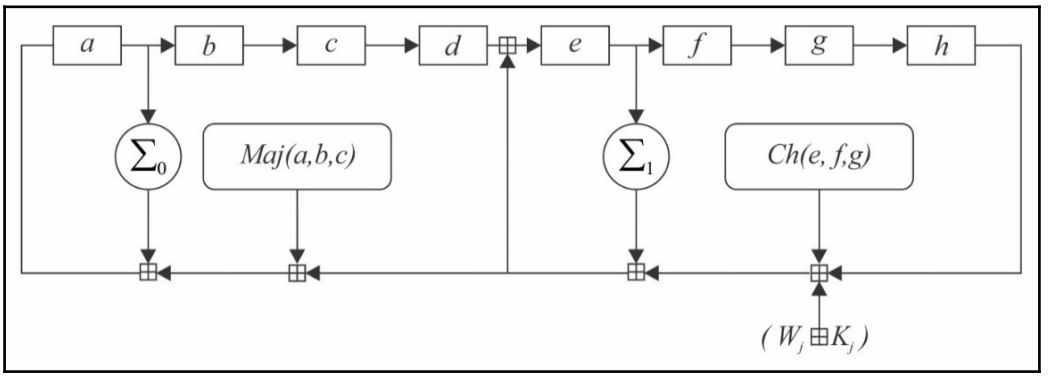
The algorithm works as follows, in eight steps:

1. **Preprocessing**:

    1. Padding of the message is used to adjust 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, which ensures that the message and its padding is divided into equal blocks of 512-bits.
    3. Setting up the initial hash value, which consists of 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 to initialize the process, and they provide a level of confidence that no backdoor exists in the algorithm.

2. **Hash computation**:

    4. Each message block is then processed in a sequence, and it requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
    5. The message schedule is prepared.
    6. Eight working variables are initialized.
    7. The intermediate hash value is calculated.
    8. Finally, the message is processed, and the output hash is produced:



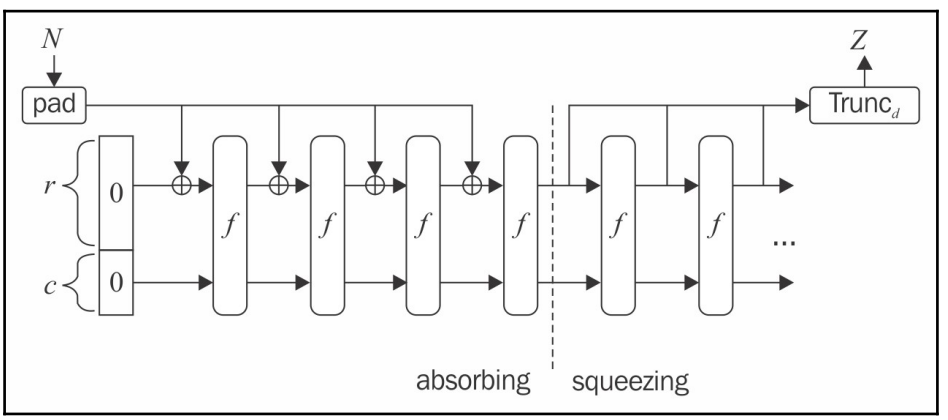
One round of a SHA-256 compression function

In the preceding diagram, **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h** are the registers. *Maj* and *Ch* are applied bitwise. $\sum_0$ and $\sum_1$ performs bitwise rotation. Round constants are $\mathbf{W}_j$ and $\mathbf{K}_j$, which are added, *mod* 2^{32}.

Design of SHA-3 (Keccak)

The structure of SHA-3 is very different from that of SHA-1 and SHA-2. The key idea behind SHA-3 is based on unkeyed permutations, as opposed to other typical hash function 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. It is a random permutation model. Different variants of SHA-3 have been standardized, such as SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256. SHAKE-128 and SHAKE-256 are **Extendable Output Functions** (**XOFs**), which are also standardized by NIST. XOFs allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model, which is the basis of SHA-3 or Keccak. Analogous to a sponge, the data is first absorbed into the sponge after applying padding. There 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. The rate is the input block size of a sponge function, while capacity determines the general security level:



SHA-3 absorbing and squeezing function

OpenSSL example of hash functions

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

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

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

```
$ 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
```

Usually, hash functions do not use a key. Nevertheless, if they *are* used with a key, then they can be used to create another cryptographic construct called MACs.

Message Authentication Codes

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

MACs using block ciphers

With this approach, block ciphers are used in the **Cipher Block Chaining** (**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 the MAC.

MACs are verified simply by computing the MAC of the message and comparing it to 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, however they cannot provide non-repudiation service due to their symmetric nature.

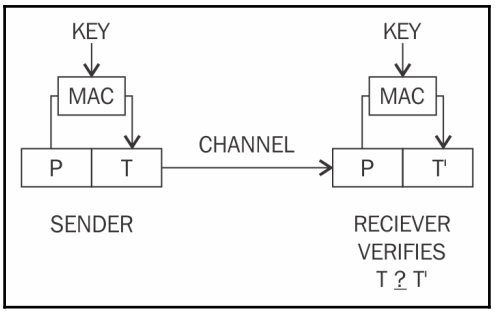
Hash-based MACs

Similar to the hash function, **Hash-based MACs** (**HMACs**) produce a fixed-length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using the 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**. With the secret prefix method, the key is concatenated with the message; that is, the key comes first and the message comes afterwards, whereas with the secret suffix method, the key comes after the message, as shown in the following equations:

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

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

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

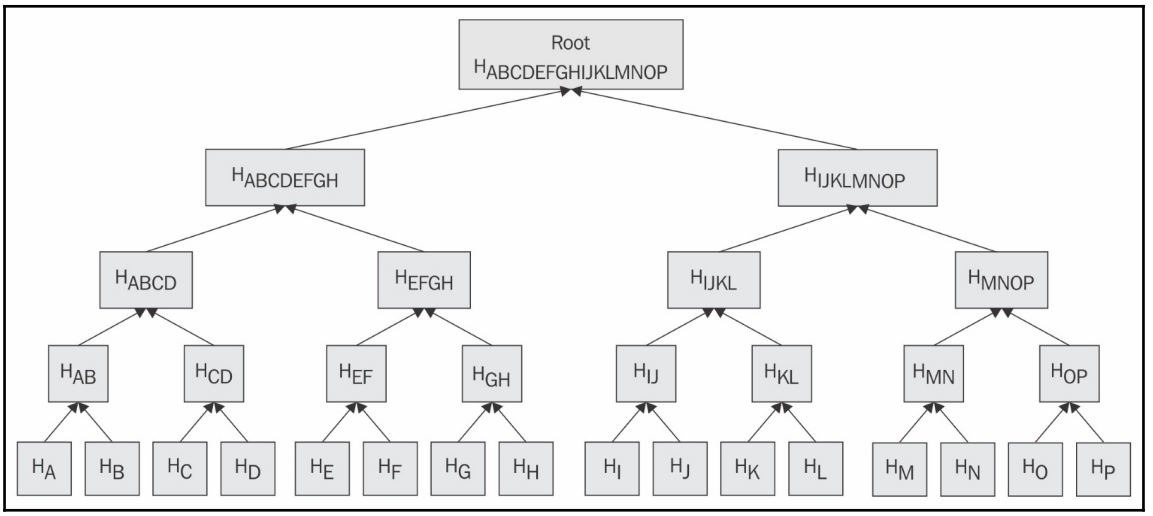


Operation of a MAC function

There are various powerful applications of hash functions used in peer-to-peer networks and blockchain technology. Some noticeable examples, such as Merkle trees, Patricia trees, and **Distributed Hash Table** (**DHT**), are discussed in the following subsections.

Merkle trees

The concept of Merkle tree was introduced by Ralph Merkle. A diagram of Merkle tree is shown here. **Merkle trees** enable secure and efficient verification of large datasets.



A Merkle tree

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

Patricia trees

To understand Patricia trees, you will first 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.

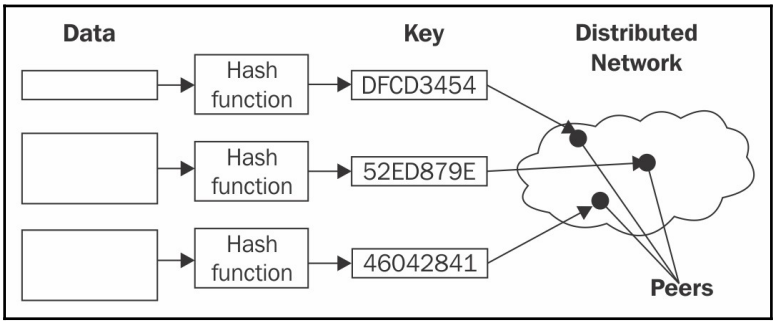
A **Merkle-Patricia tree**, based on the definitions of Patricia and Merkle, is a tree that has a root node which contains the hash value of the entire data structure.

Distributed Hash Tables

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 into a particular order.

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

The following diagram shows how a DHT works. Data is passed through a hash function, which then generates 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. DHT provides decentralization, fault tolerance, and scalability:



Distributed hash tables

Another application of hash functions is in digital signatures, where they can be used in combination with asymmetric cryptography. This concept is discussed in detail in the examples provided in the following subsections.

Digital signatures

Digital signatures provide a means of associating a message with an entity from which the message has originated. Digital signatures are used to provide data origin authentication and non-repudiation.

Digital signatures are used in blockchain where the transactions are digitally signed by senders using their private key before broadcasting the transaction to the network. This digital signing, proves they are the rightful owner of the asset, for example, bitcoins. These transactions are verified again by other nodes on the network to ensure that the funds indeed belong to the node (user) who claims to be the owner. We will discuss these concepts in more detail in chapters dedicated to Bitcoin and Ethereum in this book.

Digital signatures are calculated in two steps. As an example, the high-level steps of an RSA digital signature scheme follow.

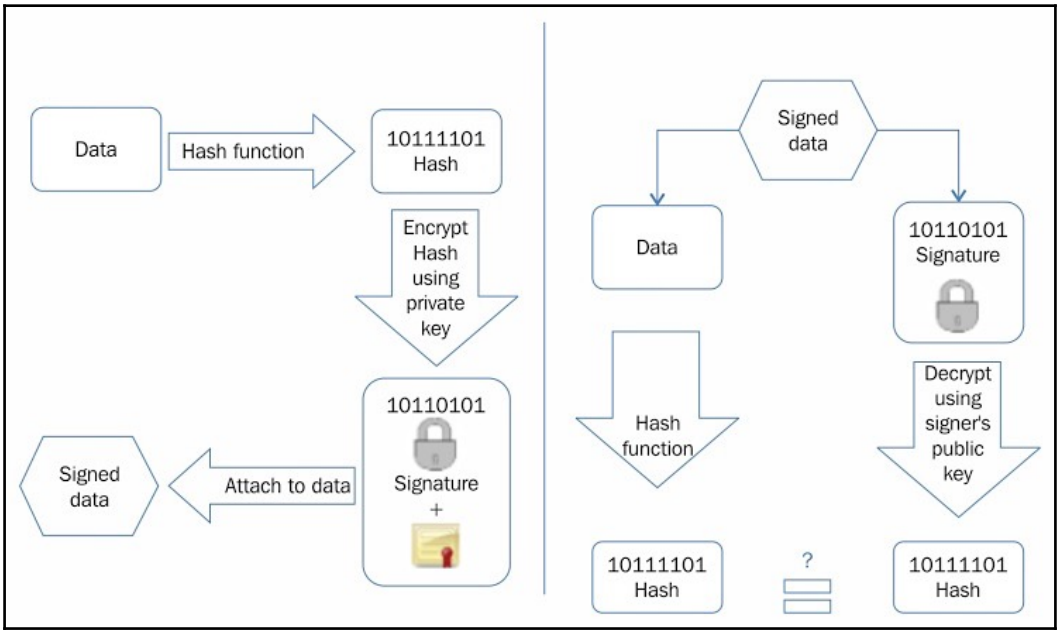
RSA digital signature algorithm

The following is the RSA digital signature algorithm:

1. **Calculate the hash value of the data packet**: This will provide the data integrity guarantee as the 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. **Signs the hash value with the signer's private key**: As only the signer 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 can use the signing functionality using the private key. In other words, no one else can produce the signed message produced by a legitimate sender. **Nonreusability** means that the digital signature cannot be separated from a message and used again for another message.

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: sign then encrypt and encrypt then sign. These two approaches to using digital signatures with encryption are as follows.

Sign then encrypt

With 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

With this method, 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** (**DSA**), and ECDSA-based digital signature schemes are used in practice. RSA is the most commonly used; however, with the traction of ECC, ECDSA-based schemes are also becoming quite popular. This is beneficial in blockchains because ECC provides same level of security that RSA does, but it uses less space. Also, generation of keys is much faster in ECC as compared to RSA, therefore it helps with overall performance of the system. The following table shows that ECC is able to provide the same level of cryptographic strength as an RSA based system with smaller key sizes:

RSA key sizes (bits)	Elliptic curve key sizes (bits)
1024	160
2048	224
3072	256
7680	384
15360	521

Comparison of RSA and Elliptic curve key sizes providing the same level of security

The ECDSA scheme is described in detail in following subsection.

Elliptic Curve Digital Signature Algorithm

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

1. First, define an elliptic curve E:
    - With modulus P
    - Coefficients a and b

- 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 $B = d\,A$.

The public key is the sextuple in the form shown here:

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

The private key, d is randomly chosen in step 2:

$$Kpr = d$$

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

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

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

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 $w = s\text{-}1 \bmod q$.
2. Auxiliary value $u1 = w.\ h(m) \bmod q$.
3. Auxiliary value $u2 = w.\ r \bmod q$.
4. Calculate point P, $P = u1A + u2B$.

5. Verification is carried out as follows:

   r, *s* is accepted as a valid signature if the *x* coordinate of point *P* calculated in step 4 has the same value as the signature parameter *r mod q*; that is:

   Xp = r mod q means valid signature

   Xp != r mod q means invalid signature

Various practical examples are shown in the following subsections, which demonstrate how the RSA digital signature can be generated, used, and verified using OpenSSL.

How to generate a digital signature using OpenSSL

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

```
$ 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:

```
$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin message.txt
```

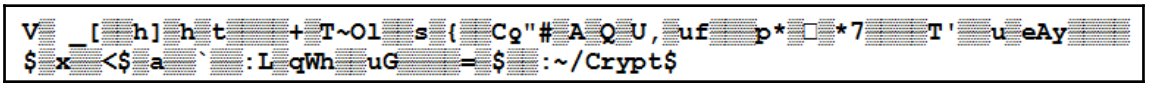
Now, let's display the directory showing the relevant files:

```
$ 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
```

Let's look at the contents of `signature.bin` by executing the following command:

```
$ cat signature.bin
```

Executing this command will give the following output:



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

```
$ openssl dgst -sha256 -verify publickey.pem -signature \
  signature.bin message.txt
Verified OK
```

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

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

Next, an example is presented 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:

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

Next, the public key is generated from the private key:

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

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

1. Create a test file:

```
$ echo testing > testsign.txt
$ cat testsign.txt
testing
```

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

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

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

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

A certificate can also be produced by using the private key generated earlier by using the following command:

```
$ openssl req -new -key eccprivatekey.pem -x509 -nodes -days 365 \
  -out ecccertificate.pem
```

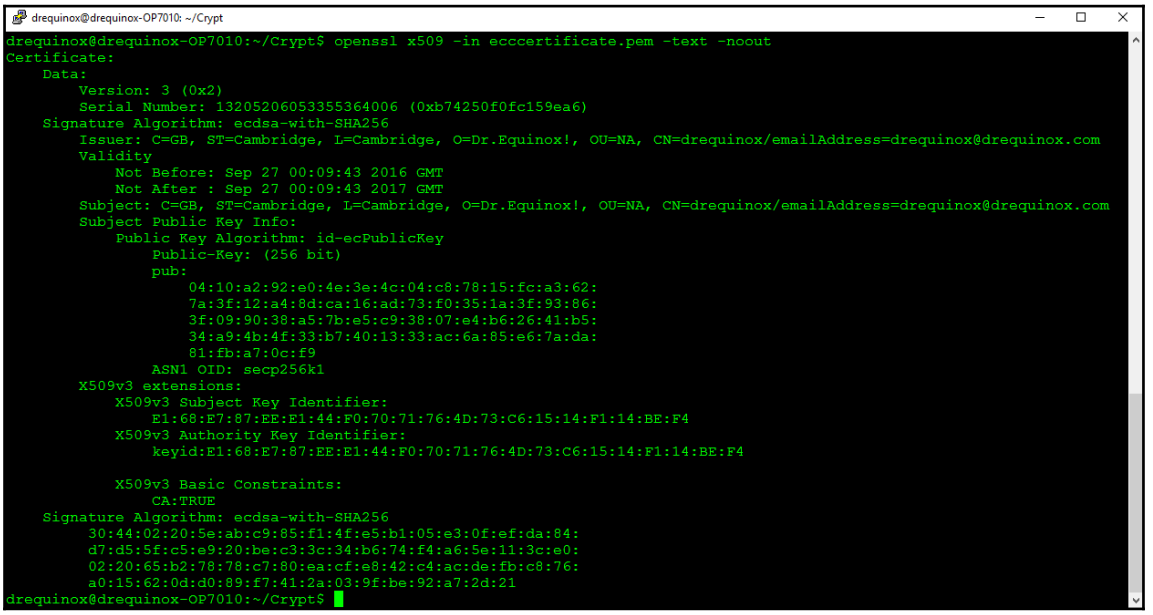
This command will produce the output similar to the one shown here. Enter the appropriate parameters to generate the certificate:

```
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 following command:

```
$ openssl x509 -in ecccertificate.pem -text -noout
```

The following output shows the certificate:



X509 certificate that uses ECDSA algorithm with SHA-256

There following topics in cryptography are presented because of their relevance to blockchain, or their potential use in future blockchain ecosystems.

Homomorphic encryption

Usually, public key cryptosystems, such as RSA, are multiplicative homomorphic or additive homomorphic, such as the Paillier cryptosystem, and are called **Partially Homomorphic Encryption** (**PHE**) 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 Encryption** (**FHE**) system was discovered by Craig Gentry. As these schemes enable the processing of encrypted data without the need for decryption, they have many different potential applications, especially in scenarios where maintaining privacy is required, but data is also mandated to be processed by potentially untrusted parties, for example, cloud computing and online search engines. Recent development in homomorphic encryption have been very promising, and researchers are actively working to make it efficient and more practical. This is of particular interest in blockchain technology, as described later in this book, as it can solve the problem of confidentiality and privacy in the blockchain.

Signcryption

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

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

Zero-Knowledge Proofs

Zero-Knowledge Proofs (**ZKPs**) were introduced by Goldwasser, Micali, and Rackoff in 1985. 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: 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. The **zero-knowledge property**, as the name implies, is the key property of ZKPs whereby it is ensured that absolutely nothing is revealed about the assertion except whether it is true or false.

ZKPs have sparked a special interest among researchers in the blockchain space due to their privacy properties, which are very much desirable in financial and many other fields, including law and medicine. A recent example of the successful implementation of a ZKP mechanism is the Zcash cryptocurrency. In Zcash, a specific type of ZKP, known as **Zero-Knowledge Succinct Non-Interactive Argument of Knowledge** (**ZK-SNARK**), is implemented. This will be discussed in detail in `Chapter 8`, *Alternative Coins*.

Blind signatures

Blind signatures were introduced by David Chaum in 1982. They 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 use is to convert binary data into text so that it can either be 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 use in Bitcoin.

Cryptography is a vast field, and this section has merely introduced the basic concepts that are essential to understanding cryptography in general and specifically from the blockchain and cryptocurrency point of view. In the next section, basic financial market concepts will be presented.

The section describes general terminology related to trading, exchanges, and the trade life cycle. More detailed information will be provided in later chapters, where specific use cases are discussed.

Financial markets and trading

Financial markets enable trading of financial securities such as bonds, equities, derivatives and currencies. There are broadly three types of markets: money markets, credit markets, and capital markets:

- **Money markets**: These 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**: These 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**: These 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 other investors via stock exchanges. Various electronic trading systems are used by exchanges today to facilitate the trading of financial instruments.

Trading

A market is a place where parties engage in exchange. It can be either a physical location or an electronic or 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, in other words, if they have sold a contract they have a short position and have a long position when they buy a contract. There are various ways to transact trades, such as through brokers or directly on an exchange or **Over-The-Counter** (**OTC**) where buyers and sellers trade directly with each other instead of using an exchange. **Brokers** are agents who arrange trades for their customers. Brokers act on a client's behalf to deal at a given price or the best possible price.

Exchanges

Exchanges are usually considered to be a very safe, regulated, and reliable place for trading. During the last decade, electronic trading has gained more popularity as compared to traditional floor-based trading. Now traders send orders to a central electronic order book from which the orders, prices, and related attributes are published to all associated systems using communications networks, thus in essence creating a virtual marketplace. Exchange trades can be performed only by members of the exchange. To trade without these limitations, the counterparties can participate in OTC trading directly.

Orders and order properties

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

- The instrument name
- Quantity to be traded
- Direction (buy or sell)
- The type of the order that represents various conditions, for example, limit orders and stop orders are orders to buy or sell once the price hits the price specified in the order, for example, Google shares for 200 GBP. Limit order allows selling or buying of stock at a specific price or better than the specified price in the order. For example, sell Microsoft shares if price is 100 USD or better.

Orders are traded by 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 delivers 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 order.

A **market order** is an instruction to trade at the best price currently available in the market. 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 to sell or buy. All of these orders are managed in an **order book**, which is a list of orders maintained by exchange, and it records the intention of buying or selling by the traders.

A **position** is a commitment to sell or buy a number of financial instruments, including 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 they come under the broad 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 of the details related to a trade. However, there is some variation depending on the type of the instrument and the asset class. These elements are described in the following subsections.

The underlying instrument

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

The attributes of financial instruments are discussed in the following subsection.

General attributes

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

Economics

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

Sales

Sales refer to the sales-characteristic related details, such as the name of the salesperson. It is just an informational field, usually without any impact on the trade life cycle.

Counterparty

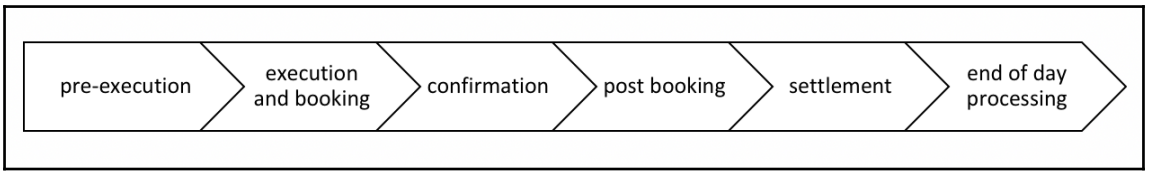
The **counterparty** is an essential component of a trade as it shows the other side (the other party involved in the trade) of the trade, and it is required to settle the trade successfully. The normal attributes include counterparty name, address, payment type, any reference IDs, settlement date, and delivery type.

Trade life cycle

A general **trade life cycle** includes the various stages from order placement to execution and 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 it into a trade. At this stage, the contract between counterparties is matured.
- **Confirmation**: This is where both counterparties agree to the particulars of the trade.
- **Post booking**: This stage is concerned with various scrutiny and verification processes required to ascertain the correctness of the trade.
- **Settlement**: This is the most vital part of trade life cycle. 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.

This life cycle is also shown in the following screenshot:



Trade life cycle

In all the aforementioned 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 to understanding the strict and necessary rules and regulations that govern the financial industry. Some concepts are described here and then again in later chapters when specific use cases are discussed. These ideas will help you understand the scenarios described.

Order anticipators

Order anticipators try to make a profit before other traders can carry out trading. This is based on the anticipation of a trader who 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 many countries. Fraudulent traders can spread false information in the market, which can then result in price movements thus enabling illegal profiteering. Usually, manipulative market conduct is trade-based, and it includes generalized and time-specific manipulations. Actions that can create an artificial shortage of stock, an impression of false activity, and price manipulation to gain criminal benefits are included in this category.

Both of the terms discussed here are relevant to the financial crime. However, there is a possibility of developing blockchain-based systems that can thwart market abuse due to its transparency and security properties.

Summary

We started this chapter with the introduction of asymmetric key cryptography. We discussed various constructs such as RSA and ECC. We also performed some experiments using OpenSSL to see that how theoretical concepts can be implemented practically. After this, we discussed hash functions in detail along with its properties and usage. Next, we covered concepts such as Merkle trees, which are used extensively in blockchain and, in fact, are at its core. We also presented other concepts such as Patricia trees and hash tables.

In the next chapter, we will present Bitcoin, which is the first blockchain invented in 2009 with the introduction of Bitcoin cryptocurrency.