CHAPTER 2

How Blockchain Works

We stand at the edge of a new digital revolution. Blockchain probably is the biggest invention since the Internet itself! It is the most promising technology for the next generation of Internet interaction systems and has received extensive attention from many industry sectors as well as academia. Today, many organizations have already realized that they needed to be blockchain ready to sustain their positions in the market. We already looked at a few use cases in Chapter 1, but the possibilities are limitless. Though blockchain is not a silver bullet for all business problems, it has started to impact most business functions and their technology implementations.

To be able to solve some real-world business problems using blockchain, we actually need a fine-grained understanding of what it is and how it works. For this, it needs to be understood through different perspectives such as business, technical, and legal viewpoints. This chapter is an effort to get into the nuts and bolts of blockchain technology and get a complete understanding of how it works.

B. Singhal et al., *Beginning Blockchain*, https://doi.org/10.1007/978-1-4842-3444-0_2

Laying the Blockchain Foundation

Blockchain is not just a technology, it is mostly coupled with business functions and use cases. In its cryptocurrency implementations, it is also interwoven with economic principles. In this section, we will mainly focus on its technical aspects. Technically, blockchain is a brilliant amalgamation of the concepts from cryptography, game theory, and computer science engineering, as shown in Figure 2-1.

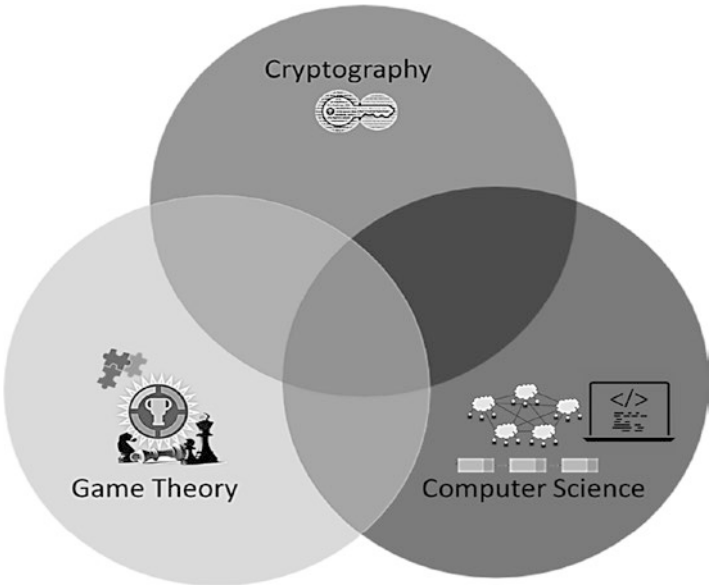


Figure 2-1. *Blockchain at its core*

Let us take a look at what role these components play in the blockchain system at a high level and dig deeper into the fundamentals eventually. Before that, let us quickly revisit how the traditional centralized systems worked. The traditional approach was that there would be a centralized entity that would maintain just one transaction/modification history. This was to exercise concurrency control over the entire database and inject

trust into the system through intermediaries. What was the problem with such a stable system then? A centralized system has to be trusted, whether those involved are honest or not! Also, cost due to intermediaries and the transaction time could be greater for obvious reasons. Now think about the centralization of power; having full control of the entire system enables the centralized authorities to do almost anything they want.

Now, let us look at how blockchain addresses these issues due to centralized intermediaries by using cryptography, game theory, and computer science concepts. Irrespective of the use case, the transactions are secured using cryptography. Using cryptography, it can be ensured that a valid user is initiating the transaction and no one can forge a fraudulent transaction. This means, cryptographically it can be ensured that Alice in no way can make a transaction on behalf of Bob by forging his signature. Now, what if a node or a user tries to launch a double-spend attack (e.g., one has just ten bucks and tries to pay the same to multiple people)? Pay close attention here–despite not having sufficient funds, one can still initiate a double-spend, which is cryptographically correct. The only way to prevent double-spend is for every node to be aware of all the transactions. Now this leads to another interesting problem. Since every node should maintain the transaction database, how can they all agree on a common database state? Again, how can the system stay immune to situations where one or more computing nodes deliberately attempt to subvert the system and try to inject a fraudulent database state? The majority of such problems come under the umbrella of the Byzantine Generals' Problem (described later). Well, it gained even more popularity because of blockchain, but it has been there for ages. If you look at the data center solutions, or distributed database solutions, the Byzantine Generals' Problem is an obvious and common problem that they deal with to remain fault tolerant. Such situations and their solution actually come from game theory. The field of game theory provides a radically different approach to determine how a system will behave. The techniques in game theory are arguably the most sophisticated and realistic ones. They

usually never consider if a node is honest, malicious, ethical, or has any other such characteristics and believe that the participants act according to the advantage they get, not by moral values. The sole purpose of game theory in blockchain is to ensure that the system is stable (i.e., in Nash Equilibrium) with consensus among the participants.

There are different kinds of business problems and situations with varying degrees of complexities. So, the underlying crypto and game theoretic consensus protocols could be different in different use cases. However, the general principle of maintaining a consistent log or database of verified transactions is the same. Though the concepts of cryptography and game theory have been around for quite some time now, it is the computer science piece that stitches these bits and pieces together through data structures and peer-to-peer network communication technique. Obviously, it is the "smart software engineering" that is needed to realize any logical or mathematical concepts in the digital world. It is then the computer science engineering techniques that incorporate cryptography and game theoretic concepts into an application, enabling decentralized and distributed computing among the nodes with data structure and network communication components.

Cryptography

Cryptography is the most important component of blockchain. It is certainly a research field in itself and is based on advanced mathematical techniques that are quite complex to understand. We will try to develop a solid understanding of some of the cryptographic concepts in this section, because different problems may require different cryptographic solutions; one size never fits all. You may skip some of the details or refer to them as and when needed, but it is the most important component to ensure security in the system. There have been many hacks reported on wallets and exchanges due to weaker design or poor cryptographic implementations.

Cryptography has been around for more than two thousand years now. It is the science of keeping things confidential using encryption techniques. However, confidentiality is not the only objective. There are various other usages of cryptography as mentioned in the following list, which we will explore later:

- **Confidentiality:** Only the intended or authorized recipient can understand the message. It can also be referred to as privacy or secrecy.
- **Data Integrity:** Data cannot be forged or modified by an adversary intentionally or by unintended/accidental errors. Though data integrity cannot prevent the alteration of data, it can provide a means of detecting whether the data was modified.
- **Authentication:** The authenticity of the sender is assured and verifiable by the receiver.
- **Non-repudiation:** The sender, after sending a message, cannot deny later that they sent the message. This means that an entity (a person or a system) cannot refuse the ownership of a previous commitment or an action.

Any information in the form of a text message, numeric data, or a computer program can be called plaintext. The idea is to encrypt the plaintext using an encryption algorithm and a key that produces the ciphertext. The ciphertext can then be transmitted to the intended recipient, who decrypts it using the decryption algorithm and the key to get the plaintext.

Let us take an example. Alice wants to send a message (m) to Bob. If she just sends the message as is, any adversary, say, Eve can easily intercept the message and the confidentiality gets compromised. So, Alice wants to encrypt the message using an encryption algorithm (E) and a

secret key (k) to produce the encrypted message called "ciphertext." An adversary has to be aware of both the algorithm (E) and key (k) to intercept the message. The stronger the algorithm and the key, the more difficult it is for the adversary to attack. Note that it would always be desirable to design blockchain systems that are at least provably secure. What this means is that a system must resist certain types of feasible attacks by adversaries.

The common set of steps for this approach can be represented as shown in Figure 2-2.

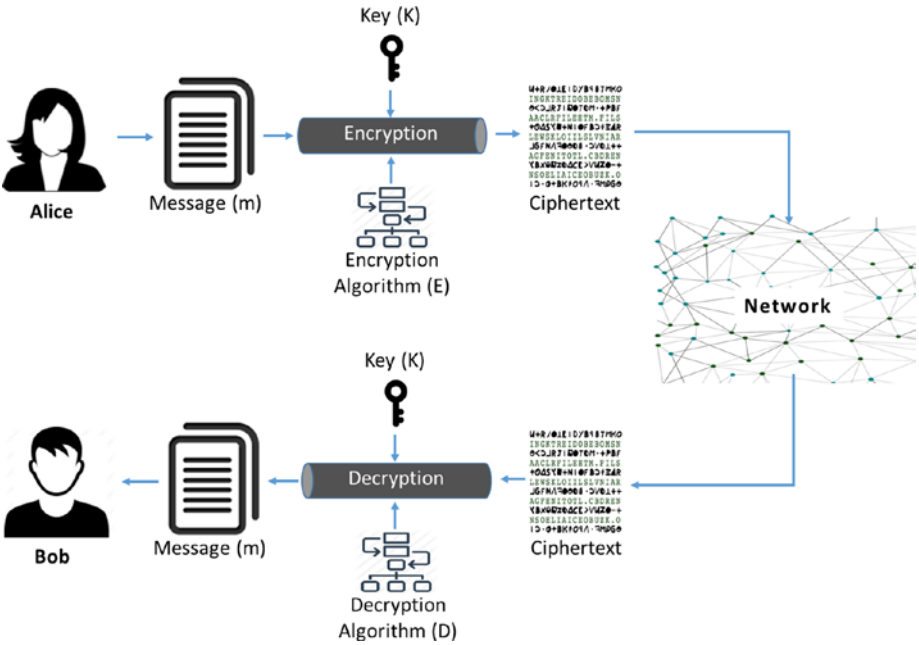


Figure 2-2. *How Cryptography works in general*

Broadly, there are two kinds of cryptography: symmetric key and asymmetric key (a.k.a. public key) cryptography. Let us look into these individually in the following sections.

Symmetric Key Cryptography

In the previous section we looked at how Alice can encrypt a message and send the ciphertext to Bob. Bob can then decrypt the ciphertext to get the original message. If the same key is used for both encryption and decryption, it is called symmetric key cryptography. This means that both Alice and Bob have to agree on a key (k) called "shared secret" before they exchange the ciphertext. So, the process is as follows:

Alice—the Sender:

- Encrypt the plaintext message m using encryption algorithm E and key k to prepare the ciphertext c
- $c = E(k, m)$
- Send the ciphertext c to Bob

Bob—the Receiver:

- Decrypt the ciphertext c using decryption algorithm D and the same key k to get the plaintext m
- $m = D(k, c)$

Did you just notice that the sender and receiver used the same key (k)? How do they agree on the same key and share it with each other? Obviously, they need a secure distribution channel to share the key. It typically looks as shown in Figure 2-3.

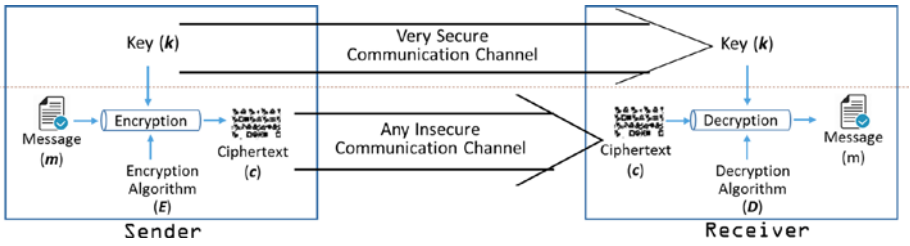


Figure 2-3. *Symmetric cryptography*

Symmetric key cryptography is used widely; the most common uses are secure file transfer protocols such as HTTPS, SFTP, and WebDAVS. Symmetric cryptosystems are usually faster and more useful when the data size is huge.

Please note that symmetric key cryptography exists in two variants: stream ciphers and block ciphers. We will discuss these in the following sections but we will look at Kerchoff's principle and XOR function before that to be able to understand how the cryptosystems really work.

Kerckhoff's Principle and XOR Function

Kerckhoff's principle states that a cryptosystem should be secured even if everything about the system is publicly known, except the key. Also, the general assumption is that the message transmission channel is never secure, and messages could easily be intercepted during transmission. This means that even if the encryption algorithm **E** and decryption algorithm **D** are public, and there is a chance that the message could be intercepted during transmission, the message is still secure due to a shared secret. So, the keys must be kept secret in a symmetric cryptosystem.

The XOR function is the basic building block for many encryption and decryption algorithms. Let us take a look at it to understand how it enables cryptography. The XOR, otherwise known as "Exclusive OR" and denoted by the symbol ⊕, can be represented by the following truth table (Table 2-1).

Table 2-1. *XOR Truth Table*

A	B	A ⊕ B
0	0	0
1	0	1
0	1	1
1	1	0

The XOR function has the following properties, which are important to understand the math behind cryptography:

- **Associative**: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- **Commutative**: $A \oplus B = B \oplus A$
- **Negation**: $A \oplus 1 = \bar{A}$
- **Identity**: $A \oplus A = 0$

Using these properties, it would now make sense how to compute the ciphertext "*c*" using plaintext "*m"* and the key "*k,"* and then decrypt the ciphertext "*c*" with the same key "*k*" to get the plaintext "*m.*" The same XOR function is used for both encryption and decryption.

$m \oplus k = c$ and $c \oplus k = m$

The previous example is in its simplest form to get the hang of encryption and decryption. Notice that it is very simple to get the original plaintext message just by XORing with the key, which is a shared secret and only known by the intended parties. Everyone may know that the encryption or decryption algorithm here is XOR, but not the key.

Stream Ciphers vs. Block Cipher

Stream cipher and block cipher algorithms differ in the way the plaintext is encoded and decoded.

Stream ciphers convert one symbol of plaintext into one symbol of ciphertext. This means that the encryption is carried out one bit or byte of plaintext at a time. In a bit by bit encryption scenario, to encrypt every bit of plaintext, a different key is generated and used. So, it uses an infinite stream of pseudorandom bits as the key and performs the XOR operation with input bits of plaintext to generate ciphertext. For such a system to remain secure, the pseudorandom keystream generator has to be secure and unpredictable. Stream ciphers are an approximation of a proven perfect cipher called "the one-time pad," which we will discuss in a little while.

How does the pseudorandom keystream get generated in the first place? They are typically generated serially from a random seed value using digital shift registers. Stream ciphers are quite simple and faster in execution. One can generate pseudorandom bits offine and decrypt very quickly, but it requires synchronization in most cases.

We saw that the pseudorandom number generator that generates the key stream is the central piece here that ensures the quality of security–which stands to be its biggest disadvantage. The pseudorandom number generator has been attacked many times in the past, which led to deprecation of stream ciphers. The most widely used stream cipher is RC4 (Rivest Cipher 4) for various protocols such as SSL, TLS, and Wi-Fi WEP/WPA etc. It was revealed that there were vulnerabilities in RC4, and it was recommended by Mozilla and Microsoft not to use it where possible.

Another disadvantage is that all information in one bit of input text is contained in its corresponding one bit of ciphertext, which is a problem of low diffusion. It could have been more secured if the information of one bit was distributed across many bits in the ciphertext output, which is the case with block ciphers. Examples of stream ciphers are one-time pad, RC4, FISH, SNOW, SEAL, A5/1, etc.

Block cipher on the other hand is based on the idea of partitioning the plaintext into relatively larger blocks of fixed-length groups of bits, and further encoding each of the blocks separately using the same key. It is a deterministic algorithm with an unvarying transformation using the symmetric key. This means when you encrypt the same plaintext block with the same key, you'll get the same result.

The usual sizes of each block are 64 bits, 128 bits, and 256 bits called block length, and their resulting ciphertext blocks are also of the same block length. We select, say, an r bits key k to encrypt every block of length n, then notice here that we have restricted the permutations of the key k to a very small subset of 2^r. This means that the notion of "perfect cipher"

does not apply. Still, random selection of the ***r*** bits secret key is important, in the sense that more randomness implies more secrecy.

To encrypt or decrypt a message in block cipher cryptography, we have to put them into a "mode of operation" that defines how to apply a cipher's single-block operation repeatedly to transform amounts of data larger than a block. Well, the mode of operation is not just to divide the data into fixed sized blocks, it has a bigger purpose. We learned that the block cipher is a deterministic algorithm. This means that the blocks with the same data, when encrypted using the same key, will produce the same ciphertext–quite dangerous! It leaks a lot of information. The idea here is to mix the plaintext blocks with the just created ciphertext blocks in some way so that for the same input blocks, their corresponding Ciphertext outputs are different. This will become clearer when we get to the DES and AES algorithms in the following sections.

Note that different modes of operations result in different properties being achieved that add to the security of the underlying block cipher. Though we will not get into the nitty-gritty of modes of operations, here are the names of a few for your reference: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR).

Block ciphers are a little slow to encrypt or decrypt, compared with the stream ciphers. Unlike stream ciphers where error propagation is much less, here the error in one bit could corrupt the whole block. On the contrary, block ciphers have the advantage of high diffusion, which means that every input plaintext bit is diffused across several ciphertext symbols. Examples of block ciphers are DES, 3DES, AES, etc.

Note A deterministic algorithm is an algorithm that, given a particular input, will always produce the same output.

One-Time Pad

It is a symmetric stream cipher where the plaintext, the key, and the ciphertext are all bit strings. Also, it is completely based on the assumption of a "purely random" key (and not pseudorandom), using which it could achieve "perfect secrecy." Also, as per the design, the key can be used only once. The problem with this is that the key should be at least as long as the plaintext. It means that if you are encrypting a 1GB file, the key would also be 1GB! This gets impractical in many real-world cases.

Example:

Table 2-2. *Example Encryption Using XOR Function*

PlainText	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
Key	0	1	0	0	1	1	0	1	1	1	0	0	1	0	1	0	1	1
Ciphertext	1	1	0	1	0	0	0	1	0	1	1	0	0	1	1	1	0	1

You can refer to the XOR truth table in the previous section to find how ciphertext is generated by XOR-ing plaintext with the key. Notice that the plaintext, the key, and the ciphertext are all 18 bits long.

Here, the receiver upon receipt of the ciphertext can simply XOR again with the key and get the plaintext. You can try it on your own with Table 2-2 and you will get the same plaintext.

The main problem with this one-time pad is more of practicality, rather than theory. How do the sender and receiver agree on a secret key that they can use? If the sender and the receiver already have a secure channel, why do they even need a key? If they do not have a secure channel (that is why we use cryptography), then how can they share the key securely? This is called the "key distribution problem."

The solution is to approximate the one-time pad by using a pseudorandom number generator (PRNG). This is a deterministic algorithm that uses a seed value to generate a sequence of random numbers that are not truly random; this in itself is an issue. The sender and the receiver have to have the same seed value for this system to work. Sharing that seed value is way better compared with sharing the entire key; just that it has to be secured. It is susceptible to compromise by someone who knows the algorithm as well as the seed.

Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric block cipher technique. It uses 64-bit block size with a 64-bit key for encryption and decryption. Out of the 64-bit key, 8 bits are reserved for parity checks and technically 56 bits is the key length. It has been proven that it is vulnerable to brute force attack and could be broken in less than a day. Given Moore's law, it could be broken a lot quicker in the future, so its usage has been deprecated quite a bit because of the key length. It was very popular and was being used in banking applications, ATMs, and other commercial applications, and more so in hardware implementations than software. We give a high-level description of the DES encryption in this section.

In symmetric cryptography, a large number of block ciphers use a design scheme known as a "Feistel cipher" or "Feistel network." A Feistel cipher consists of multiple rounds to process the plaintext with the key, and every round consists of a substitution step followed by a permutation step. The more the number of rounds, the more secure it could be but encryption/decryption gets slower. The DES is based on a Feistel cipher with 16 rounds. A general sequence of steps in the DES algorithm is shown in Figure 2-4.

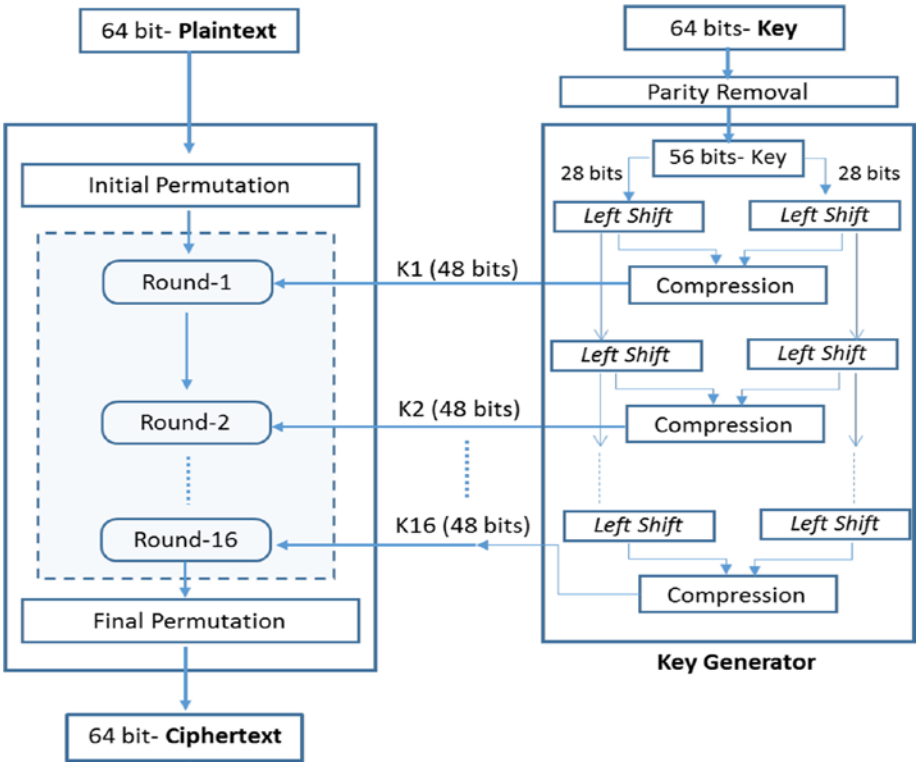


Figure 2-4. *DES cryptography*

Let us first talk about the key generator and then we will get into the encryption part.

- As mentioned before, the key is also 64 bits long. Since 8 bits are used as parity bits (more precisely, bit number 8, 16, 24, 32, 40, 48, 56, and 64), only 56 bits are used for encryption and decryption.
- After parity removal, the 56-bit key is divided into two blocks, each of 28 bits. They are then bit-wise left shifted in every round. We know that the DES uses 16 rounds of Feistel network. Note here that every round

takes the previous round's left-shifted bit block and then again left shifts by one bit in the current round.

- Both the left-shifted 28-bit blocks are then combined through a compression mechanism that outputs a 48-bit key called subkey that gets used for encryption. Similarly, in every round, the two 28-bit blocks from the previous round get left shifted again by one bit and then clubbed and compressed to the 48-bit key. This key is then fed to the encryption function of the same round.

Let us now look at how DES uses the Feistel cipher rounds for encryption:

- First, the plaintext input is divided into 64 bit blocks. If the number of bits in the message is not evenly divisible by 64, then the last block is padded to make it a 64-bit block.
- Every 64-bit input data block goes through an initial permutation (IP) round. It simply permutes, i.e., rearranges all the 64-bit inputs in a specific pattern by transposing the input blocks. It has no cryptographic significance as such, and its objective is to make it easier to load plaintext/ciphertext into DES chips in byte-sized format.
- After the IP round, the 64-bit block gets divided into two 32-bit blocks, a left block (L) and a right block (R). In every round, the blocks are represented as L_i and R_i, where the subscript "I" denotes the round. So, the outcomes of IP round are denoted as L_0 and R_0.

- Now the Feistel rounds start. The first round takes L_0 and R_0 as input and follows the following steps:
    - The right side 32-bit block (R) comes as is to the left side and the left side 32-bit block (L) goes through an operation with the key ***k*** of that round and the right side 32-bit block (R) as shown following:
    - $L_i = R_i - 1$
    - $R_i = L_i - 1 \oplus F(R_i - 1, K_i)$ where "I" is the round number
    - The F() is called the "Cipher Function" that is actually the core part of every round. There are multiple steps or operations that are bundled together in this F() operation.
    - In the first step, operation of the 32-bit R-block is expanded and permuted to output a 48-bit block.
    - In the second step, this 48-bit block is then XORed with the 48-bit subkey supplied by the key generator of the same round.
    - In the third step, this 48-bit XORed output is fed to the substitution box to reduce the bits back to 32 bits. The substitution operation in this S-box is the only nonlinear operation in DES and contributes significantly to the security of this algorithm.
    - In the fourth step, the 32-bit output of the S-box is fed to the permutation box (P-box), which is just a permutation operation that outputs a 32-bit block, which is actually the final output of F() cipher function.
    - The output of F() is then XORed with the 32-bit L-block, which is input to this round. This XORed output then becomes the final R-block output of this round.

- Refer to Figure 2-5 to understand the various operations that take place in every round.

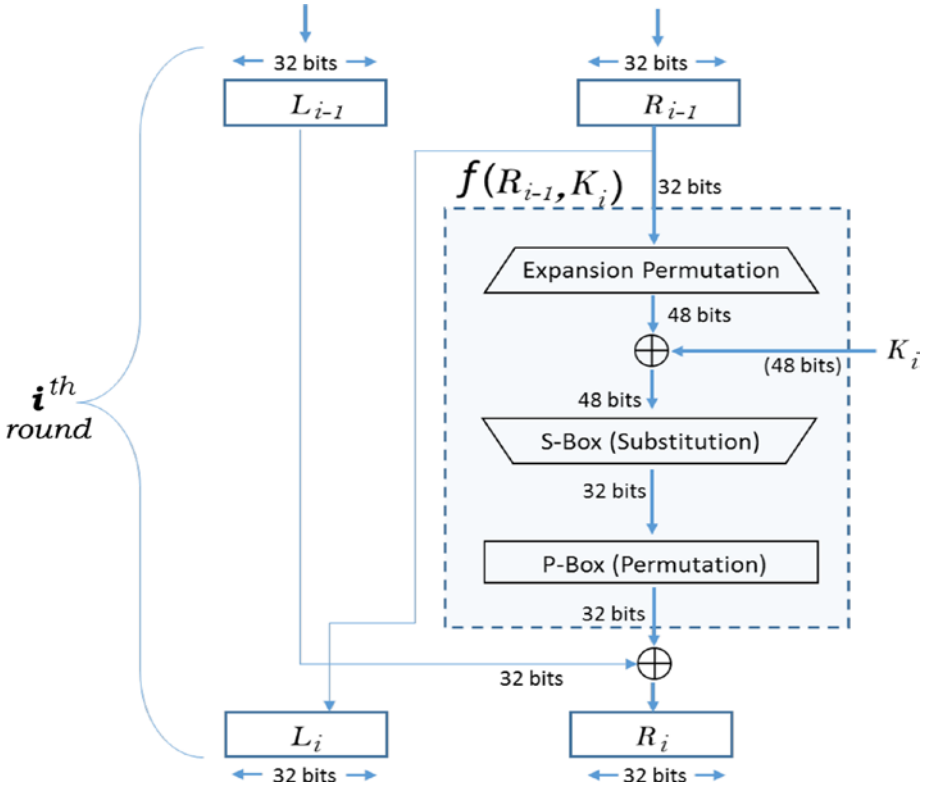


Figure 2-5. *Round function of DES*

- The previously discussed Feistel round gets repeated 16 times, where the output of one round is fed as the input to the following round.
- Once all the 16 rounds are over, the output of the 16th round is again swapped such that the left becomes the right block and vice versa.
- Then the two blocks are clubbed to make a 64-bit block and passed through a permutation operation, which is the inverse of the initial permutation function and that results in the 64-bit ciphertext output.

We looked at how the DES algorithm really works. The decryption also works a similar way in the reverse order. We will not get into those details, but leave it to you to explore.

Let us conclude with the limitations of the DES. The 56-bit key length was susceptible to brute force attack and the S-boxes used for substitution in each round were also prone to cryptanalysis attack because of some inherent weaknesses. Because of these reasons, the Advanced Encryption Standard (AES) has replaced the DES to the extent possible. Many applications now choose AES over DES.

Advanced Encryption Standard

Like DES, the AES algorithm is also a symmetric block cipher but is not based on a Feistel network. The AES uses a substitution-permutation network in a more general sense. It not only offers greater security, but also offers greater speed! As per the AES standards, the block size is fixed at 128 bits and allows a choice of three keys: 128 bits, 192 bits, and 256 bits. Depending on the choice of the key, AES is named as AES-128, AES-192, and AES-256.

In AES, the number of encryption rounds depend on the key length. For AES-128, there are ten rounds; for AES-192, there are 12 rounds; and for AES-256, there are 14 rounds. In this section, our discussion is limited to key length 128 (i.e., AES-128), as the process is almost the same for other variants of AES. The only thing that changes is the "key schedule," which we will look into later in this section.

Unlike DES, AES encryption rounds are iterative and operate an entire data block of 128 bits in every round. Also, unlike DES, the decryption is not very similar to the encryption process in AES.

To understand the processing steps in every round, consider the 128-bit block as 16 bytes where individual bytes are arranged in a 4 × 4 matrix as shown:

Byte 0	Byte 4	Byte 8	Byte 12
Byte 1	Byte 5	Byte 9	Byte 13
Byte 2	Byte 6	Byte 10	Byte 14
Byte 3	Byte 7	Byte 11	Byte 15

This 4 × 4 matrix of bytes as shown is referred to as **state array**. Please note that every round consumes an input state array and produces an output state array.

The AES also uses another piece of jargon called "word" that needs to be defined before we go further. Whereas a byte consists of eight bits, a word consists of four bytes, that is, 32 bits. The four bytes in each column of the state array form 32-bit words and can be called **state words**. The state array can be shown as follows:

$word_0$	$word_1$	$word_2$	$word_3$
Byte 0	Byte 4	Byte 8	Byte 12
Byte 1	Byte 5	Byte 9	Byte 13
Byte 2	Byte 6	Byte 10	Byte 14
Byte 3	Byte 7	Byte 11	Byte 15

Also, every byte can be represented with two hexadecimal numbers. Example: if the 8-bit byte is {00111010}, it could be represented as "3A" in Hex notation. "3" represents the left four bits "0011" and "A" represents the right four bits "1010."

Now to generalize each round, processing in each round happens at byte level and consists of byte-level substitution followed by word-level permutation, hence it is a substitution-permutation network. We will get to further details when we discuss the various operations in each round. The overall encryption and decryption process of AES can be represented in Figure 2-6.

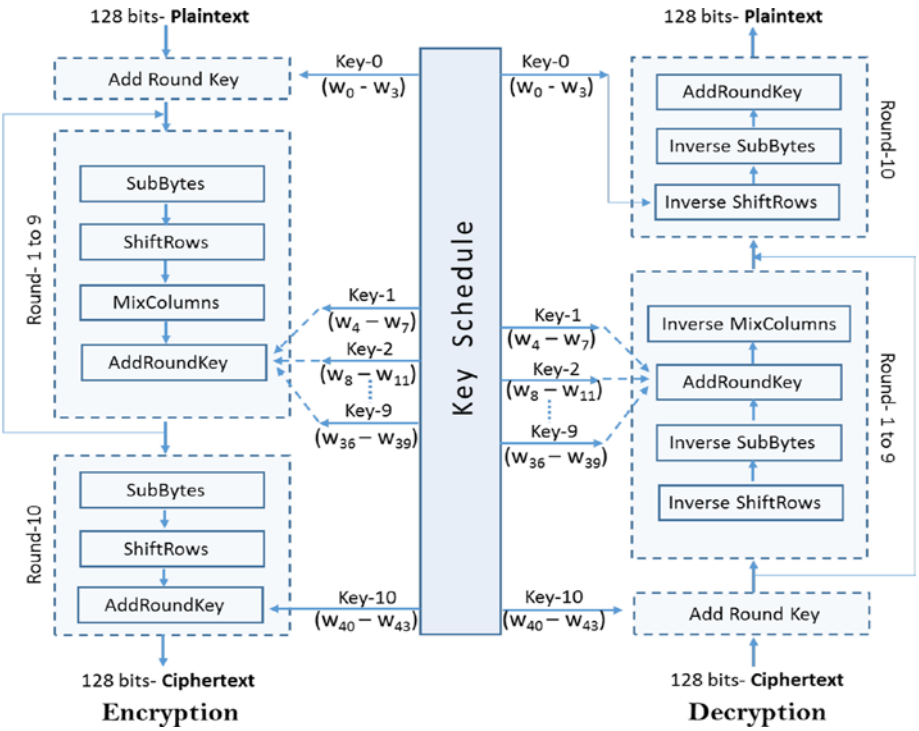


Figure 2-6. *AES cryptography*

If you paid close attention to Figure 2-6, you would have noticed that the decryption process is not just the opposite of encryption. The operations in the rounds are executed in a different order! All steps of the round function–SubBytes, ShiftRows, MixColumns, AddRoundKey–are invertible. Also, notice that the rounds are iterative in nature. Round 1 through round 9 have all four operations, and the last round excludes only the "MixColumns" operation. Let us now build a high-level understanding of each operation that takes place in a round function.

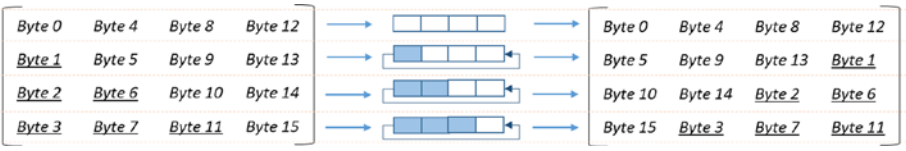
SubBytes: This is a substitution step. Here, each byte is represented as two hexadecimal digits. As an example, take a byte {00111010} represented with two hexadecimal digits, say {3A}. To find its substitution values, refer to the S-box lookup table (16 × 16 table) to find the corresponding value

for 3-row and A-column. So, for {3A}, the corresponding substituted value would be {80}. This step provides the nonlinearity in the cipher.

ShiftRows: This is the transformation step and is based upon the matrix representation of the state array. It consists of the following shift operations:

- No circular shifting of the first row, and remains as is
- Circularly shifting of the second row by one byte to the left
- Circularly shifting of the third row by two bytes to the left
- Circularly shifting of the fourth row (last row) by three bytes to the left

It can be represented as shown:



MixColumns: It is also a transformation step where all the four columns of the state are multiplied with a fixed polynomial (C_x) and get transformed to new columns. In this process, each byte of a column is mapped to a new value that is a function of all four bytes in the column. This is achieved by the matrix multiplication of state as shown:

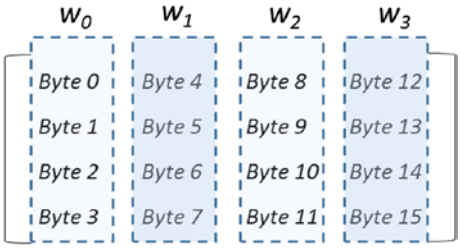
$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} Byte\ 0 & Byte\ 4 & Byte\ 8 & Byte\ 12 \\ Byte\ 1 & Byte\ 5 & Byte\ 9 & Byte\ 13 \\ Byte\ 2 & Byte\ 6 & Byte\ 10 & Byte\ 14 \\ Byte\ 3 & Byte\ 7 & Byte\ 11 & Byte\ 15 \end{bmatrix} = \begin{bmatrix} Byte\ 0' & Byte\ 4' & Byte\ 8' & Byte\ 12' \\ Byte\ 1' & Byte\ 5' & Byte\ 9' & Byte\ 13' \\ Byte\ 2' & Byte\ 6' & Byte\ 10' & Byte\ 14' \\ Byte\ 3' & Byte\ 7' & Byte\ 11' & Byte\ 15' \end{bmatrix}$$

The matrix multiplication is as usual, but the AND products are XORed. Let us see one of the examples to understand the process. Byte 0' is calculated as shown:

Byte 0' = (2 . Byte0) ⊕ (3 . Byte1) ⊕ Byte3 ⊕ Byte4

It is important to note that this MixColumns step, along with the ShiftRows step, provide the necessary diffusion property (information from one byte gets diffused to multiple bytes) to the cipher.

AddRoundKey: This is again a transformation step where the 128-bit round key is bitwise XORed with 128 bits of state in a column major order. So, the operation takes place column-wise, meaning four bytes of a word state column with one word of the round key. In the same way we represented the 128-bit plaintext block, the 128-bit key should also be represented in the same 4 × 4 matrix as shown here:



128-bit key

This operation affects every bit of a state. Now, recollect that there are ten rounds, and each round has its own round key. Since there is an "AddRoundKey" step before the rounds start, effectively there are eleven (10 + 1) AddRoundKey operations. In one round, all 128-bits of subkey, that is, all four words of subkey, are used to XOR with the 128-bit input data block. So, in total, we require 44 key words, $\mathbf{w_0}$ through $\mathbf{w_{43}}$. This is why the 128-bit key has to go through a key expansion operation, which we will get to in a little while.

Note here that the key word [$\mathbf{w_0}$, $\mathbf{w_1}$, $\mathbf{w_2}$, $\mathbf{w_3}$] gets XORed with the initial input block before the round-based processing begins. The remaining 40 word-keys, $\mathbf{w_4}$ through $\mathbf{w_{43}}$, get used four words at a time in each of the ten rounds.

AES Key Expansion: The AES key expansion algorithm takes as input a 128-bit cipher key (four-word key) and produces a schedule of 44 key words from it. The idea is to design this system in such a way that a one-bit change in the key would significantly affect all the round keys.

The key expansion operation is designed such that each grouping of a four-word key produces the next grouping of a four-word key in a four-word to four-word basis. It is easy to explain this with a pictorial representation, so here we go:

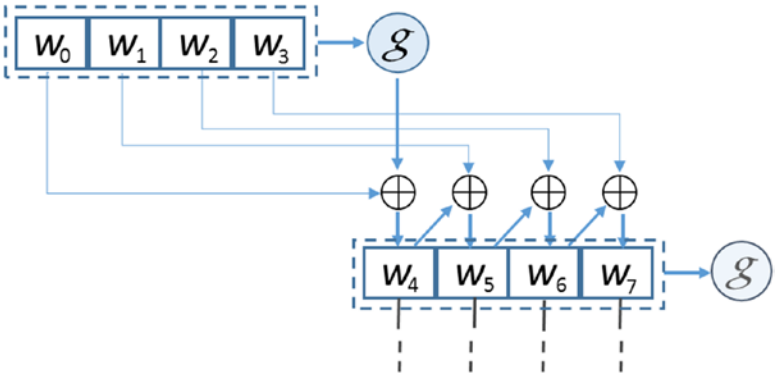


Figure 2-7. *AES key expansion*

We will quickly run through the operations that take place for key expansion by referring to the diagram:

- The initial 128-bit key is [$\mathbf{w_0}$, $\mathbf{w_1}$, $\mathbf{w_2}$, $\mathbf{w_3}$] arranged in four words.
- Take a look at the expanded word now: [$\mathbf{w_4}$, $\mathbf{w_5}$, $\mathbf{w_6}$, $\mathbf{w_7}$]. Notice that $\mathbf{w_5}$ depends on $\mathbf{w_4}$ and $\mathbf{w_1}$. This means that every expanded word depends on the immediately preceding word, i.e., $\mathbf{w_i}$ – 1 and the word that is four positions back, i.e., $\mathbf{w_i}$ – 4 . Test the same for $\mathbf{w_6}$ as well. As you can see, just a simple XOR operation is performed here.

- Now, what about $\mathbf{w_4}$? Or, any other position that is a multiple of four, such as $\mathbf{w_8}$ or $\mathbf{w_{12}}$? For these words, a more complex function denoted as "**g**" is used. It is basically a three-step function. In the first step, the input four-word block goes through circular left shift by one byte. For example [$\mathbf{w_0}$, $\mathbf{w_1}$, $\mathbf{w_2}$, $\mathbf{w_3}$] becomes [$\mathbf{w_1}$, $\mathbf{w_2}$, $\mathbf{w_3}$, $\mathbf{w_0}$]. In the second step, the four bytes input word (e.g., [$\mathbf{w_1}$, $\mathbf{w_2}$, $\mathbf{w_3}$, $\mathbf{w_0}$]) is taken as input and byte substitution is applied on each byte using S-box. Then, in the third step, the result of step 2 is XORed with something called round constant denoted as Rcon[ ]. The round constant is a word in which the right-most three bytes are always zero. For example, [x, 0, 0, 0]. This means that the purpose of Rcon[ ] is to just perform XOR on the left-most byte of the step 2 output key word. Also note that the Rcon[ ] is different for each round. This way, the final output of the complex function "**g**" is generated, which is then XORed with $\mathbf{w_i}$ – 4 to get $\mathbf{w_i}$ where "I" is a multiple of 4.
- This is how the key expansion takes place in AES.

The output state array of the last round is rearranged back to form the 128-bit ciphertext block. Similarly, the decryption process takes place in a different order, which we looked at in the AES process diagram. The idea was to give you a heads-up on how this algorithm works at a high level, and we will restrict our discussion to just the encryption process in this section.

The AES algorithm is standardized by the NIST (National Institute of Standards and Technology). It had the limitation of long processing time. Assume that you are sending just a 1 megabyte file (8388608 bits) by encrypting with AES. Using a 128-bit AES algorithm, the number of steps required for this encryption will be 8388608/128 = 65536 on this

same number of data blocks! Using a parallel processing approach, AES efficiency can be increased, but is still not very suitable when you are dealing with large data.

Challenges in Symmetric Key Cryptography

There are some limitations in symmetric key cryptography. A few of them are listed as follows:

- The key must be shared by the sender and receiver before any communication. It requires a secured key establishment mechanism in place.
- The sender and receiver must trust each other, as they use the same symmetric key. If a receiver is hacked by an attacker or the receiver deliberately shared the key with someone else, the system gets compromised.
- A large network of, say, n nodes require key $n(n–1)/2$ key pairs to be managed.
- It is advisable to keep changing the key for each communication session.
- Often a trusted third party is needed for effective key management, which itself is a big issue.

Cryptographic Hash Functions

Hash functions are the mathematical functions that are the most important cryptographic primitives and are an integral part of blockchain data structure. They are widely used in many cryptographic protocols, information security applications such as Digital Signatures and message authentication codes (MACs). Since it is used in asymmetric key cryptography, we will discuss it here prior to getting into asymmetric

cryptography. Please note that the concepts covered in this section may not be in accordance with the academic text books, and a little biased toward the blockchain ecosystem.

Cryptographic hash functions are a special class of hash functions that are apt for cryptography, and we will limit our discussion pertaining to it only. So, a cryptographic hash function is a one-way function that converts input data of arbitrary length and produces a fixed-length output. The output is usually termed "hash value" or "message digest." It can be represented as shown Figure 2-8.

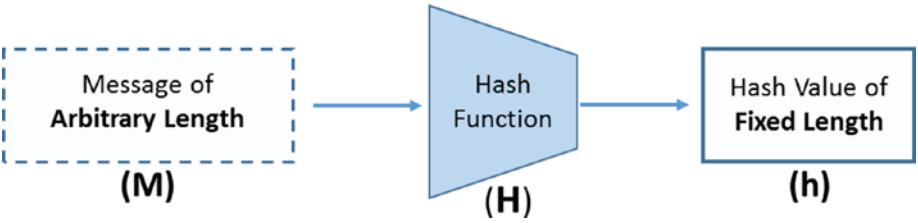


Figure 2-8. *Hash function in its basic form*

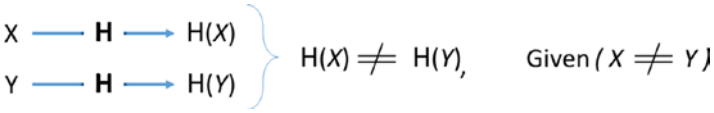
For the hash functions to serve their design purpose and be usable, they should have the following core properties:

- Input can be any string of any size, but the output is of fixed length, say, a 256-bit output or a 512-bit output as examples.
- The hash value should be efficiently computable for any given message.
- It is deterministic, in the sense that the same input when provided to the same hash function produces the same hash value every time.
- It is infeasible (though not impossible!) to invert and generate the message from its hash value, except trying for all possible messages.

- Any small change in the message should greatly influence the output hash, just so no one can correlate the new hash value with the old one after a small change.

Apart from the aforementioned core properties, they should also meet the following security properties to be considered as a cryptographic protocol:

- Collision resistance: It implies that it is infeasible to find two different inputs, say, X and Y, that hash to the same value.



  This makes the hash function H() collision resistant because no one can find X and Y, such that H(X) = H(Y). Note that this hash function is a compression function, as it compresses a given input to fixed sized output that is shorter than the input. So, the input space is too large (anything of any size) compared with the output space, which is fixed. If the output is a 256-bit hash value, then the output space can have a maximum of 2^{256} values, and not beyond that. This implies that a collision must exist. However, it is extremely difficult to find that collision. As per the theory of "the birthday paradox," we can infer that it should be possible to find a collision by using the square root of the output space. So, by taking $2^{128} + 1$ inputs, it is highly likely to find a collision; but that is an extremely huge number to compute, which is quite infeasible!

Let us now discuss where this property could be useful. In the majority of online storage, cloud file storage, blob storage, App Stores, etc., the property "collision resistance" is widely used to ensure the integrity of the files. Example: someone computes the message digest of a file and uploads to cloud storage. Later when they download the file, they could just compute the message digest again and cross-check with the old one they have. This way, it can be ensured if the file was corrupted because of some transmission issues or possibly due to some deliberate attempts. It is due to the property of collision resistance that no one can come up with a different file or a modified file that would hash to the same value as that of the original file.

- Preimage resistance: This property means that it is computationally impossible to invert a hash function; i.e., finding the input X from the output H(X) is infeasible. Therefore, this property can also be called "hiding" property. Pay close attention here; there is another subtle aspect to this situation. Note that when X can be anything in the world, this property is easily achieved. However, if there are just a limited number of values that X can take, and that is known to the adversary, they can easily compute all possible values of X and find which one hashes to the outcome.

  Example: A laboratory decided to prepare the message digests for the successful outcome of an experiment so that any adversary who gets access to the results database cannot make any sense of it because what is stored in the system are hashed outputs. Assume that there can only be three possible outcomes of the

experiment such as OP111, OP112, and OP113, out of which only one is successful, say, OP112. So, the laboratory decides to hash it, compute H(OP112), and store the hashed values in the system. Though an adversary cannot find OP112 from H(OP112), they can simply hash all the possible outcomes of the experiment, i.e., H(OP111), H(OP112), and H(OP113) and see that only H(OP112) is matching with what is stored in the system. Such a situation is certainly vulnerable! This means that, when the input to a hash function comes from a limited space and does not come from a spread-out distribution, it is weak. However, there is a solution to it as follows:

Let us take an input, say "**X**" that is not very spread out, just like the outcomes of the experiment we just discussed with a few possible values. If we can concatenate that with another random input, say "**r**," that comes from a probability distribution with high min entropy, then it will be difficult to find X from H(**r** || **X**). Here, high min entropy means that the distribution is very spread out and there is no particular value that is likely to occur. Assume that "**r**" was chosen from 256-bit distribution. For an adversary to get the exact value of "**r**" that was used along with input, there is a success probability of $1/2^{256}$, which is almost impossible to achieve. The only way is to consider all the possible values of this distribution one by one—which is again practically impossible. The value "**r**" is also referred to as "nonce." In cryptography, a nonce is a random number that can be used only once.

Let us now discuss where this property of preimage resistance could be useful. It is very useful in committing to a value, so "commitment" is the use case here. This can be better explained with an example. Assume that you have participated in some sort of betting or gambling event. Say you have to commit to your option, and declare it as well. However, no one should be able to figure out what you are betting on, and you yourself cannot deny later on what you bet on. So, you leverage the preimage resistance property of Hash Function. You take a hash of the choice you are betting on, and declare it publicly. No one can invert the hash function and figure out what you are betting on. Also, you cannot later say that your choice was different, because if you hash a different choice, it will not match what you have declared publicly. It is advisable to use a nonce "**r**" the way we explained in the previous paragraph to design such systems.

- Second preimage resistance: This property is slightly different from "collision resistant." It implies that given an input X and its hash H(X), it is infeasible to find Y, such that H(X) = H(Y). Unlike in collision-resistant property, here the discussion is for a given X, which is fixed. This implies that if a hash function is collision resistant already, then it is second preimage resistant also.

There is another derived property from the properties mentioned that is quite useful in Bitcoin. Let us look into it from a technical point of view and learn how Bitcoin leverages it for mining when we hit Chapter 3. The name of this property is "puzzle friendliness." This name implies that there is no shortcut to the solution and the only way to get to the solution is to

traverse through all the possible options in the input space. We will not try to define it here but will directly try to understand what it really means. Let us consider this example: H(**r** || **X**) = **Z**, where "**r**" is chosen from a distribution with high min entropy, "**X**" is the input concatenated with "**r**," and "**Z**" is the hashed output value. The property means that it is way too hard for an adversary to find a value "**Y**" that exactly hashes to "**Z**." That is, H(**r′** || **Y**) = **Z**, where **r′** is a part of the input chosen in the same randomized way as "**r**." What this means is that, when a part of the input is substantially randomized, it is hard to break the hash function with a quick solution; the only way is to test with all possible random values.

In the previous example, if "**Z**" is an **n**-bits output, then it has taken just one value out of 2^n possible values. Note carefully that a part of your input, say "**r**," is from a high min-entropy distribution, which has to be appended with your input **X**. Now comes the interesting part of designing a search puzzle. Let's say **Z** is an **n**-bits output and is a set of 2^n possible values, not just an exact value. You are asked to find a value of **r** such that when hashed appended with **X**, it falls within that output set of 2^n values; then it forms a search puzzle. The idea is to find all possible values of **r** till it falls withing the range of **Z**. Note here that the size of **Z** has limited the output space to a smaller set of 2^n possible values. The smaller the output space, the harder is the problem. Obviously, if the range is big, it is easier to find a value in it and if the range is quite narrow with just a few possibilities, then finding a value within that range is tough. This is the beauty of the "**r**," called the "nonce" in the input to hash function. Whatever random value of **r** you take, it will be concatenated with "**X**" and will go through the same hash function, again and again, till you get the right nonce value "**r**" that satisfies the required range for **Z**, and there are absolutely no shortcuts to it except for trying all possible values!

Note that for an **n**-bit hash value output, an average effort of 2^n is needed to break preimage and second preimage resistance, and $2^n/2$ for collision resistance.

We discussed various fundamental and security properties of hash functions. In the following sections we will see some important hash functions and dive deeper as applicable.

A Heads-up on Different Hash Functions

One of the oldest hash functions or compression function is the MD4 hash function. It belongs to the message digest (MD) family. Other members of the MD family are MD5 and MD6, and there are many other variants of MD4 such as RIPEMD. The MD family of algorithms produce a 128-bit message digest by consuming 512-bit blocks. They were widely used as checksums to verify data integrity. Many file servers or software repositories used to provide a precomputed MD5 checksum, which the users could check against the file they downloaded. However, there were a lot of vulnerabilities found in the MD family and it was deprecated.

Another such hash function family is the Secure Hash Algorithm (SHA) family. There are basically four algorithms in this family, such as SHA-0, SHA-1, SHA-2, and SHA-3. The first algorithm proposed in this family was named SHA, but newer versions were coming with security fixes and updates, so a retronym was applied to it and it was made SHA-0. It was found to have a serious yet undisclosed security flaw and was discontinued. Later, SHA-1 was proposed as a replacement to SHA-0. SHA-1 had an extra computational step that addressed the problem in SHA-0. Both SHA-0 and SHA-1 were 160-bit hash functions that consumed 512-bit block sizes. SHA-1 was designed by the National Security Agency (NSA) to use it in the digital signature algorithm (DSA). It was used quite a lot in many security tools and Internet protocols such as SSL, SSH, TSL, etc. It was also used in version control systems such as Mercurial, Git, etc. for consistency checks, and not really for security. Later, around 2005, cryptographic weaknesses were found in it and it was deprecated after the year 2010. We will get into SHA-2 and SHA-3 in detail in the following sections.

SHA-2

It belongs to the SHA family of hash functions, but itself is a family of hash functions. It has many SHA variants such as SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. SHA-256 and SHA-512 are the primitive hash functions and the other variants are derived from them. The SHA-2 family of hash functions are widely used in applications such as SSL, SSH, TSL, PGP, MIME, etc.

SHA-224 is a truncated version of SHA-256 with a different initial value or initialization vector (IV). Note that the SHA variants with different truncations applied can produce the same bit length hash outputs, hence different initialization vectors are applied in different SHA variants to be able to properly differentiate them. Now coming back to the SHA-224 computation, it is a two-step process. First, SHA-256 value is computed with a different IV compared with the default one used in SHA-256. Second, the resulting 256-bit hash value is truncated to 224-bit; usually the 224 bits from left are kept, but the choice is all yours.

SHA-384 is a truncated version of SHA-512, just the way SHA-224 is a truncated version of SHA-256. Similarly, both 512/224 and SHA-512/256 are truncated versions of SHA-512. Are you wondering why this concept of "truncation" exists? Note that truncation is not just limited to the ones we just mentioned, and there can be various other variants as well. The primary reasons for truncation could be as follows:

- Some applications require a message digest with a certain length that is different from the default ones.
- Irrespective of the SHA-2 variant we are using, we can select a truncation level depending on what security property we want to sustain. Example: Considering today's state of computing power, when collision resistance is necessary, we should keep at least 160 bits and when only preimage-resistance is necessary,

we should keep at least 80 bits. The security property such as collision resistance decreases with truncation, but it should be chosen such that it would be computationally infeasible to find a collision.

- Truncation also helps maintain the backward compatibility with older applications. Example: SHA-224 provides 112-bit security that can match the key length of triple-DES (3DES).

Talking about efficiency, SHA-256 is based on a 32-bit word and SHA-512 is based on a 64-bit word. So, on a 64-bit architecture, SHA-512 and all its truncated variants can be computed faster with a better level of security compared with SHA-1 or other SHA-256 variants.

Table 2-3 is a tabular representation taken from the NIST paper that represents SHA-1 and different SHA-2 algorithms properties in a nutshell.

Table 2-3. *SHA-1 & SHA-2 Hash Function in a Nutshell*

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

As a rule of thumb, it is advisable not to truncate when not necessary. Certain hash functions tolerate truncation and some don't, and it also depends on how you are using it and in what context.

SHA-256 and SHA-512

As mentioned already, SHA-256 belongs to the SHA-2 family of hash functions, and this is the one used in Bitcoins! As the name suggests, it produces a 256-bit hash value, hence the name. So, it can provide 2^{128}-bit security as per the birthday paradox.

Recall that the hash functions take arbitrary length input and produce a fixed size output. The arbitrary length input is not fed as is to the compression function and is broken into fixed length blocks before it is fed to the compression function. This means that a construction method is needed that can iterate through the compression function by constructing fixed-sized input blocks from arbitrary length input data and produce a fixed length output. There are various types of construction methods such as Merkle-Damgård construction, tree construction, and sponge construction. It is proven that if the underlying compression function is collision resistant, then the overall hash function with any construction method should also be collision resistant.

The construction method that SHA-256 uses is the Merkle-Damgård construction, so let us see how it works in Figure 2-9.

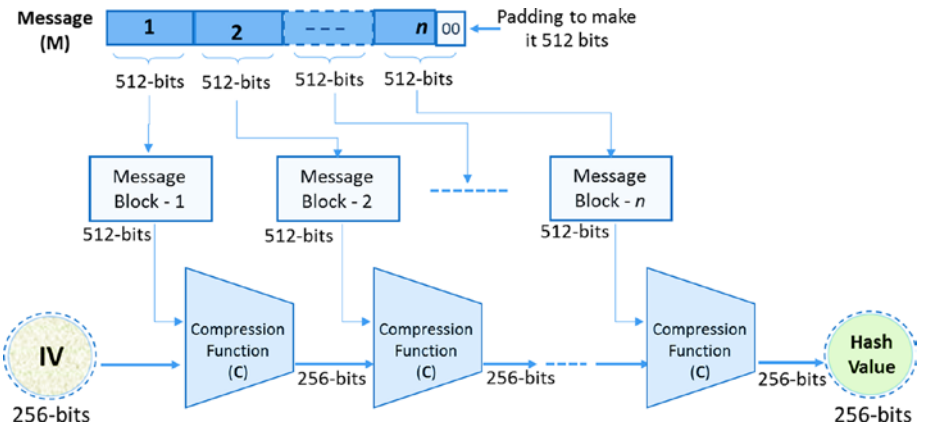


Figure 2-9. *Merkle-Damgård construction for SHA-256*

Referring to the diagram, the following steps (presented at a high level) are executed in the order specified to compute the final hash value:

- As you can see in the diagram, the message is first divided into 512-bit blocks. When the message is not an exact multiple of 512 bits (which is usually the case), the last block falls short of bits, hence it is padded to make it 512 bits.
- The 512-bit blocks are further divided into 16 blocks of 32-bit words (16 × 32 = 512).
- Each block goes through 64 rounds of round function where each 32-bit word goes through a series of operations. The round functions are a combination of some common functions such as XOR, AND, OR, NOT, Bit-wise Left/Right Shift, etc. and we will not get into those details in this book.

Similar to SHA-256, the steps and the operations are quite similar in SHA-512, as SHA-512 also uses Merkle-Damgård construction. The major difference is that there are 80 rounds of round functions in SHA-512 and the word length is 64 bits. The block size in SHA-512 is 1024 bits, which gets further divided into 16 blocks of 64-bit words The output message digest is 512 bits in length, that is, eight blocks of 64-bit words. While SHA-512 was gaining momentum, and started being used in many applications, a few people turned to the SHA-3 algorithm to be future ready. SHA-3 is just a different approach to hashing and not a real replacement to SHA-256 or SHA-512, though it allows tuning. We will learn a few more details about SHA-3 in the following sections.

RIPEMD

RACE Integrity Primitives Evaluation Message Digest (RIPEMD) hash function is a variant of the MD4 hash function with almost the same design considerations. Since it is used in Bitcoins, we will have a brief discussion on it.

The original RIPEMD was of 128 bits, later RIPEMD-160 was developed. There exist 128-, 256-, and 320-bit versions of this algorithm, called RIPEMD-128, RIPEMD-256, and RIPEMD-320, respectively, but we will limit our discussion to the most popular and widely used RIPEMD-160.

RIPEMD-160 is a cryptographic hash function whose compression function is based on the Merkle–Damgård construction. The input is broken into 512-bit blocks and padding is applied when the input bits are not a multiple of 512. The 160-bit hash value output is usually represented as 40-digit hexadecimal numbers.

The compression function is made up of 80 stages, made up of two parallel lines of five rounds of 16 steps each ($5 \times 16 = 80$). The compression function works on sixteen 32-bit words (512-bit blocks).

Note Bitcoin uses both SHA-256 and RIPEMD-160 hashes together for address generation. RIPEMD-160 is used to further shorten the hash value output of SHA-256 to 160 bits.

SHA-3

In 2015, the Keccak (pronounced as “ket-chak”) algorithm was standardized by the NIST as the SHA-3. Note that the purpose was not really to replace the SHA-2 standard, but to complement and coexist with it, though one can choose SHA-3 over SHA-2 in some situations.

Since both SHA-1 and SHA-2 were based on Merkle-Damgård construction, a different approach to hash function was desirable. So, not using Merkle-Damgård construction was one of the criteria set by the NIST. This was because the new design should not suffer from the limitations of Merkle-Damgård construction such as multicollision. Keccak, which became SHA-3, used a different construction method called sponge construction.

In order to make it backward compatible, it was required that SHA-3 should be able to produce variable length outputs such as 224, 256, 384, and 512 bits and also other arbitrary length outputs. This way SHA-3 became a family of cryptographic hash functions such as SHA3-224, SHA3-256, SHA3-384, SHA3 -512, and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256. Also, SHA-3 had to have a tunable parameter (capacity) to allow a tradeoff between security and performance. Since SHAKE128 and SHAKE256 are XOFs, their output can be extended to any desired length, hence the name.

The following diagram (Figure 2-10) shows how SHA-3 (Keccak algorithm) is designed at a high level.

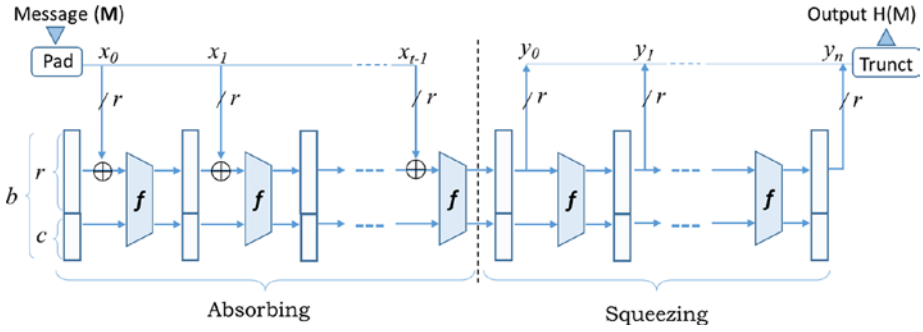


Figure 2-10. *Sponge construction for SHA-3*

A series of steps that take place for SHA-3 are as follows:

- As you can see in Figure 2-10, the message is first divided into blocks ($\mathbf{x_i}$) of size ***r*** bits. If the input data is not a multiple of ***r*** bits, then padding is required. If you are wondering about this **r**, do not worry, we will get to it in a little while. Now, let us focus on how this padding happens. For a message block $\mathbf{x_i}$ which is not a multiple of **r** and has some message **m** in it, padding happens as shown in the following:

  $\mathbf{x_i}$ = m || P 1 {0}* 1

  "P" is a predetermined bit string followed by 1 {0}* 1, which means a leading and trailing 1 and some number of zeros (could be no zero bits also) that can make $\mathbf{x_i}$ a multiple of **r**. Table 2-4 shows the various values of P.

Table 2-4. *Padding in SHA-3 variants*

Mode	Output Length	P	1 {0}* 1
SHA3-224	224	11001	1 {0}* 1
SHA3-256	256	11101	1 {0}* 1
SHA3-384	384	11001	1 {0}* 1
SHA3 -512	512	11101	1 {0}* 1
Variable Length (XOFs)	Arbitrary	1111	1 {0}* 1

- As you can see in Figure 2-10, there are broadly two phases to SHA-3 sponge construction: the first one is the "Absorbing" phase for input, and the second one is the "Squeezing" phase for output. In the Absorbing phase, the message blocks ($\mathbf{x_i}$) go through various operations of the algorithm and in the Squeezing

phase, the output of configurable length is computed. Notice that for both of these phases, the same function called "Kecaak-f" is used.

- For the computation of SHA3-224, SHA3-256, SHA3-384, SHA3 -512, which is effectively a replacement of SHA-2, only the first bits of the first output block y_0 are used with required level of truncation.
- The SHA-3 is designed to be tunable for its security strength, input, and output sizes with the help of tuning parameters.
- As you can see in the diagram, "b" represents the width of the state and requires that r + c = b. Also, "b" depends on the exponent "ℓ" such that $b = 25 \times 2^{\ell}$
- Since "ℓ" can take on values between 0 and 6, "b"can have widths {25, 50, 100, 200, 400, 800 and 1600}. It is advisable not to use the smallest two values of "b" in practice as they are just there to analyze and perform cryptanalysis on the algorithm.
- In the equation r + c = b, the "r" that we see is what we used to preprocess the message and divided into blocks of length "r." This is called the "bit rate." Also, the parameter "c" is called the capacity that just has to satisfy the condition $r + c = b \in \{25, 50, 100, 200, 400, 800, 1600\}$ and get computed. This way "r" and "c" are used as tuning parameters to trade off between security and performance.
- For SHA-3, the exponent value ℓ is fixed to be "6," so the value of b is 1600 bits. For this given b = 1600, two bit-rate values are permissible: r = 1344 and r = 1088. This leads to two distinct values of "c." So, for r = 1344, c = 256 and for r = 1088, c = 512.

- Let us now look at the core engine of this algorithm, i.e. Keccak-*f*, which is also called "Keccak-*f* Permutation." There are "n" rounds in each Keccak-*f*, where "n" is computed as: $n = 12 + 2\ell$. Since the value of ℓ is 6 for SHA-3, there will be 24 rounds in each Keccak-f. Every round takes "b" bits (r + c) input and produces the same number of "b" bits as output.
- In each round, the input "b" is called a state. This state array "b" can be represented as a three-dimensional (3-D) array b = (5 x 5 × w), where word size $w = 2^{\ell}$. So, w = 64 bits, which means 5 × 5 = 25 words of 64 bits each. Recall that $\ell = 6$ for SHA-3, so b = 5 × 5 x 64 = 1600. The 3-D array can be shown as in Figure 2-11.

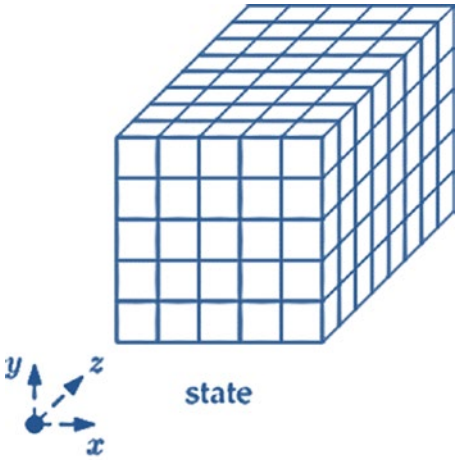


Figure 2-11. *State array representationin SHA-3*

- Each round consists of a sequence of five steps and the state array gets manipulated in each of those steps as shown in Figure 2-12.

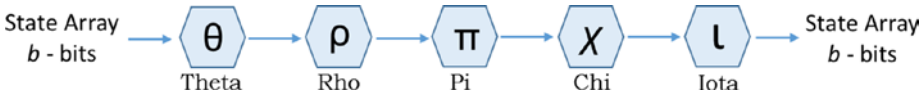


Figure 2-12. *The five steps in each SHA-3 round*

- Without getting into much detail into each of the five steps, let us quickly learn what they do at a high level:
  - Theta (θ) step: It performs the XOR operation to provide minor diffusion.
  - Rho (ρ) step: It performs bitwise rotation of each of the 25 words.
  - Pi (π) step: It performs permutation of each of the 25 words.
  - Chi (χ) step: In this step, bits are replaced by combining those with their two subsequent bits in their rows.
  - Iota (ι) step: It XORs a round constant into one word of the state to break the symmetry.
- The last round of Keccak-*f* produces the y_0 output, which is enough for SHA-2 replacement mode, i.e., the output with 224, 256, 384, and 512 bits. Note that the least significant bits of y_0 are used for the desired length output. In case of variable length output, along with y_0, other output bits of y_1, y_2, y_3… can also be used.

When it comes to the real-life implementation of SHA-3, it is found that its performance is good in software (though not as good as SHA-2) and is excellent in hardware (better than SHA-2).

Applications of Hash Functions

The cryptographic hash functions have many different usages in different situations. Following are a few example use cases:

- Hash functions are used in verifying the integrity and authenticity of information.
- Hash functions can also be used to index data in hash tables. This can speed up the process of searching. Instead of the whole data, if we search based on the hashes (assuming the much shorter hash value compared with the whole data), then it should obviously be faster.
- They can be used to securely authenticate the users without storing the passwords locally. Imagine a situation where you do not want to store passwords on the server, obviously because if an adversary hacks on to the server, they cannot get the password from their stored hashes. Every time a user tries to log in, hash of the punched in password is calculated and matched against the stored hash. Secured, isn't it?
- Since hash functions are one-way functions, they can be used to implement PRNG.
- Bitcoin uses hash functions as a proof of work (PoW) algorithm. We will get into the details of it when we hit the Bitcoin chapter.

- Bitcoin also uses hash functions to generate addresses to improve security and privacy.
- The two most important applications are digital signatures and in MACs such as hash-based message authentication codes (HMACs).

Understanding the working and the properties of the hash functions, there can be various other use cases where hash functions can be used.

Note The Internet Engineering Task Force (IETF) adopted version 3.0 of the SSL (SSLv3) protocol in 1999, renamed it to Transport Layer Security (TLS) version 1.0 (TLSv1) protocol and defined it in RFC 2246. SSLv3 and TLSv1 are compatible as far as the basic operations are concerned.

Code Examples of Hash Functions

Following are some code examples of different hash functions. This section is just intended to give you a heads-up on how to use the hash functions programatically. Code examples are in Python but would be quite similar in different languages; you just have to find the right library functions to use.

```
# -*- coding: utf-8 -*-
import hashlib
# hashlib module is a popular module to do hashing in python

#Constructors of md5(), sha1(), sha224(), sha256(), sha384(),
and sha512() present in hashlib
md=hashlib.md5()
md.update("The quick brown fox jumps over the lazy dog")
print md.digest()
```

```
print "Digest Size:", md.digest_size, "\n", "Block Size: ",
md.block_size

# Comparing digest of SHA224, SHA256,SHA384,SHA512
print "Digest SHA224", hashlib.sha224("The quick brown fox
jumps over the lazy dog").hexdigest()
print "Digest SHA256", hashlib.sha256("The quick brown fox
jumps over the lazy dog").hexdigest()
print "Digest SHA384", hashlib.sha384("The quick brown fox
jumps over the lazy dog").hexdigest()
print "Digest SHA512", hashlib.sha512("The quick brown fox
jumps over the lazy dog").hexdigest()
# All hashoutputs are unique

# RIPEMD160 160 bit hashing example
h = hashlib.new('ripemd160')
h.update("The quick brown fox jumps over the lazy dog")
h.hexdigest()

#Key derivation Alogithm:
#Native hashing algorithms are not resistant against brutefore
attack.
#Key deviation algorithms are used for securing password
hashing.
import hashlib, binascii
algorithm='sha256'
password='HomeWifi'
salt='salt' # salt is random data that can be used as an
additional input to a one-way function
nu_rounds=1000
key_length=64 #dklen is the length of the derived key
dk = hashlib.pbkdf2_hmac(algorithm,password, salt, nu_rounds,
dklen=key_length)
```

```
print 'derieved key: ',dk
print 'derieved key in hexadeximal :', binascii.hexlify(dk)

# Check properties for hash
import hashlib

input = "Sample Input Text"
for i in xrange(20):
    # add the iterator to the end of the text
    input_text = input + str(i)
    # show the input and hash result
    print input_text, ':',  hashlib.sha256(input_text).
    hexdigest()
```

MAC and HMAC

HMAC is a type of MAC (message authentication code). As the name suggests, a MAC's purpose is to provide message authentication using Symmetric Key and message integrity using hash functions. So, the sender sends the MAC along with the message for the receiver to verify and trust it. The receiver already has the key *K* (as symmetric key cryptography is being used, so both sender and receiver have agreed on it already); they just use it to compute the MAC of the message and check it against the MAC that was sent along with the message.

In its simplest form, MAC = H(key || message). HMAC is actually a technique to turn the hash functions into MACs. In HMAC, the hash functions can be applied multiple times along with the key and its derived keys. HMACs are widely used in RFID-based systems, TLS, etc. In SSL/TLS (HTTPS is TTP within SSL/TLS), HMAC is used to allow client and server to verify and ensure that the exchanged data has not been altered during transmission. Let us take a look at a few of the important MAC strategies that are widely used:

- **MAC-then-Encrypt**: This technique requires the computation of MAC on the cleartext, appending it to the data, and then encrypting all of that together. This scheme does not provide integrity of the ciphertext. At the receiving end, the message decryption has to happen first to be able to check the integrity of the message. It ensures the integrity of the plaintext, however. TLS uses this scheme of MAC to ensure that the client-server communication session is secured.
- **Encrypt-and-MAC**: This technique requires the encryption and MAC computation of the message or the cleartext, and then appending the MAC at the end of the encrypted message or ciphertext. Notice that MAC is computed on the cleartext, so integrity of the cleartext can be assured but not of the ciphertext, which leaves scope for some attacks. Unlike the previous scheme, integrity of the cleartext can be verified. SSH (Secure Shell)uses this MAC scheme.
- **Encrypt-then-MAC**: This technique requires that the cleartext needs to be encrypted first, and then compute the MAC on the ciphertext. This MAC of the ciphertext is then appended to the ciphertext itself. This scheme ensures integrity of the ciphertext, so it is possible to first check the integrity and if valid then decrypt it. It easily filters out the invalid ciphertexts, which makes it efficient in many cases. Also, since MAC is in ciphertext, in no way does it reveal information about the plaintext. It is usually the most ideal of all schemes and has wider implementations. It is used in IPsec.

Asymmetric Key Cryptography

Asymmetric key cryptography, also known as "public key cryptography," is a revolutionary concept introduced by Diffie and Hellman. With this technique, they solved the problem of key distribution in a symmetric cryptography system by introducing digital signatures. Note that asymmetric key cryptography does not eliminate the need for symmetric key cryptography. They usually complement each other; the advantages of one can compensate for the disadvantages of the other.

Let us see a practical scenario to understand how such a system would work. Assume that Alice wants to send a message to Bob confidentially so that no one other than Bob can make sense of the message, then it would require the following steps:

Alice—The Sender:

- Encrypt the plaintext message **m** using encryption algorithm **E** and the public key $\mathbf{Puk_{Bob}}$ to prepare the ciphertext **c**.
- $\mathbf{c} = \mathbf{E}(\mathbf{Puk_{Bob}}, \mathbf{m})$
- Send the ciphertext c to Bob.

Bob—The Receiver:

- Decrypt the ciphertext **c** using decryption algorithm **D** and its private key $\mathbf{Prk_{Bob}}$ to get the original plaintext **m**.
- $\mathbf{m} = \mathbf{D}(\mathbf{Prk_{Bob}}, \mathbf{c})$

Such a system can be represented as shown in Figure 2-13.

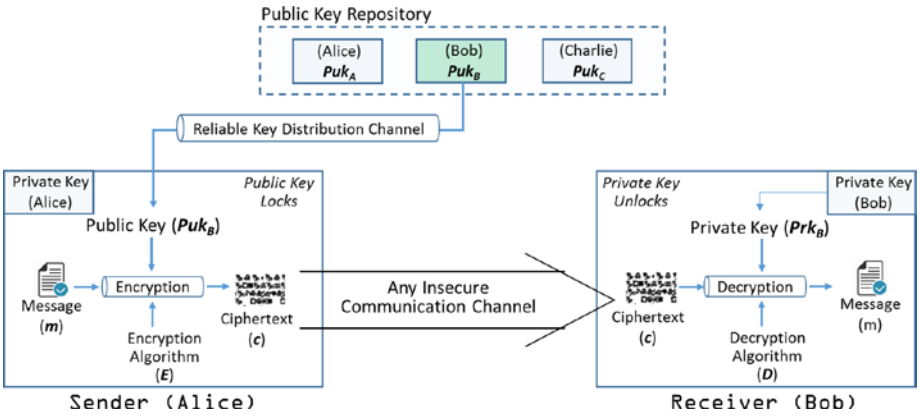


Figure 2-13. *Asymmetric cryptography for confidentiality*

Notice that the public key should be kept in a public repository accessible to everyone and the private key should be kept as a well-guarded secret. Public key cryptography also provides a way of authentication. The receiver, Bob, can verify the authenticity of the origin of the message m in the same way.

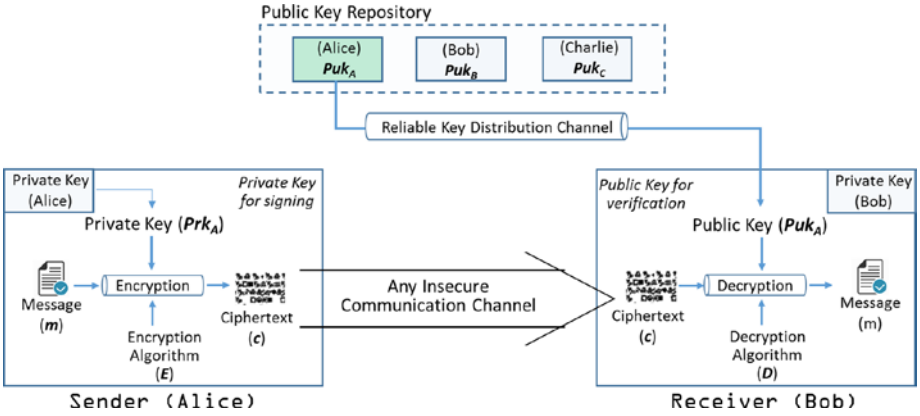


Figure 2-14. *Asymmetric cryptography for authentication*

In the example in Figure 2-14, the message was prepared using Alice's private key, so it could be ensured that it only came from Alice. So, the entire message served as a digital signature. Note that both confidentiality and authentication are desirable. To facilitate this, public key encryption has to be used twice. The message should first be encrypted with the sender's private key to provide a digital signature. Then it should be encrypted with the receiver's public key to provide confidentiality. It can be represented as:

- $\mathbf{c = E[Puk_{Bob}, E(Prk_{Alice}, m)]}$
- $\mathbf{m = D[Puk_{Alice}, D(Prk_{Bob}, c)]}$

As you can see, the decryption happens in just its reverse order. Notice that the public key cryptography is used four times here: twice for encryption and twice for decryption. It is also possible that the sender may sign the message by applying the private key to just a small block of data derived from the message to be sent, and not to the whole message. In the real world, App stores such as Google Play or Apple App Store require that the software apps should be digitally signed before they get published.

We looked at the uses of the two keys in asymmetric cryptography, which can be summarized as follows:

- Public keys are known and accessible to everyone. They can be used to encrypt the message or to verify the signatures.
- Private keys are extremely private to individuals. They are used to decrypt the message or to create signatures.

In asymmetric or public key cryptography, there is no key distribution problem, as exchanging the agreed upon key is no longer needed. However, there is a significant challenge with this approach. How would one ensure that the public key they are using to encrypt the message is really the public key of the intended recipient and not of an intruder or eavesdropper? To solve this, the notion of a trusted third party called public

key infrastructure (PKI) is introduced. Through PKIs, the authenticity of public keys is assured by the process of attestation or notarization of user identity. The way PKIs operate is that they provide verified public keys by embedding them in a security certificate by digitally signing them.

The public key encryption scheme can also be called one-way function or a trapdoor function. This is because encrypting a plaintext using the public key "Puk" is easy, but the other direction is practically impossible. No one really can deduce the original plaintext from the encrypted ciphertext without knowing the secret or private key "Prk," which is actually the trapdoor information. Also, in the context of just the keys, they are mathematically related but it is computationally not feasible to find one from the other.

We discussed the important objectives of public key cryptography such as key establishment, authentication and non-repudiation through digital signatures, and confidentiality through encryption. However, not all public key cryptography algorithms may provide all these three characteristics. Also, the algorithms are different in terms of their underlying computational problem and are classified accordingly. Certain algorithms such as RSA are based on integer factorization scheme because it is difficult to factor large numbers. Certain algorithms are based on the discrete logarithm problems in finite fields such as Diffie–Hellman key exchange (DH) and DSA. A generalized version of discrete logarithm problems is elliptic curve (EC) public key schemes. The Elliptic Curve Digital Signature Algorithm (ECDSA) is an example of it. We will cover most of these algorithms in the following section.

RSA

RSA algorithm, named after Ron Rivest, Adi Shamir, and Leonard Adleman is possibly one of the most widely used cryptographic algorithms. It is based on the practical difficulty of factoring very large numbers. In RSA, plaintext and ciphertext are integers between **0** and $\mathbf{n-1}$ for some **n**.

We will discuss the RSA scheme from two aspects. First is generation of key pairs and second, how the encryption and decryption works. Since modular arithmetic provides the mechanism for key generation, let us quickly look at it.

Modular Arithmetic

Let **m** be a positive integer called modulus. Two integers **a** and **b** are congruent modulo **m** if:

a ≡ **b** (mod **m**), which implies **a** − **b** = **m . k** for some integer **k**.

Example: if **a** ≡ 16 (mod 10) then **a** can have the following solutions:

a = . . ., −24, − 14, −4, 6, 16, 26, 36, 46

Any of these numbers subtracted by 16 is divisible by 10. For example, −24 −16 = −40, which is divisible by 10. Note that **a** ≡ 36 (mod 10) can also have the same solutions of **a**.

As per the Quotient-Remainder theorem, only a unique solution of "**a**" exists that satisfies the condition: **0** ≤ **a** < **m**. In the example **a** ≡ 16 (mod 10), only the value 6 satisfies the condition 0 ≤ 6 < 10. This is what will be used in the encryption/decryption process of RSA algorithm.

Let us now look at the Inverse Midulus. If **b** is an inverse to **a** modulo **m**, then it can be represented as:

a b ≡ **1** (mod **m**), which implies that **a b** − **1** = **m . k** for some integer **k**.

Example: 3 has inverse 7 modulo 10 since

3 · 7 = 1 (mod 10) => 21 − 1 = 20, which is divisible by 10.

Generation of Key Pairs

As discussed already, a key pair of private and public keys is needed for any party to participate in asymmetric crypto-communication. In the RSA scheme, the public key consists of (**e**, **n**) where **n** is called the modulus and **e** is called the public exponent. Similarly, the private key consists of (**d**, **n**), where **n** is the same modulus and **d** is the private exponent.

Let us see how these keys get generated along with an example:

- Generate a pair of two large prime numbers **p** and **q**. Let us take two small prime numbers as an example here for the sake of easy understanding. So, let the two primes be **p** = 7 and **q** = 17.
- Compute the RSA modulus (**n**) as **n** = **pq**. This **n** should be a large number, typically a minimum of 512 bits. In our example, the modulus (**n**) = **pq** = 119.
- Find a public exponent **e** such that **1** < **e** < (**p** – **1**) (**q** – **1**) and there must be no common factor for **e** and (**p** – **1**) (**q** – **1**) except **1**. It implies that **e** and (**p** – **1**) (**q** – **1**) are coprime. Note that there can be multiple values that satisfy this condition and can be taken as **e**, but any one should be taken.
- In our example, (**p** – 1) (**q** – 1) = 6 × 16 = 96. So, **e** can be relatively prime to and less than 96. Let us take **e** to be 5.
- Now the pair of numbers (**e**, **n**) form the public key and should be made public. So, in our example, the public key is (5, 119).
- Calculate the private exponent **d** using **p**, **q**, and **e** considering the number **d** is the inverse of **e** modulo (**p** – 1) (**q** – 1). This implies that **d** when multiplied by **e** is equal to **1** modulo (**p** – **1**) (**q** – **1**) and **d** < (**p** – **1**) (**q** – **1**). It can be represented as:

  e d = **1** mod (**p** – **1**) (**q** – **1**)
- Note that this multiplicative inverse is the link between the private key and the public key. Though the keys are not derived from each other, there is a relation between them.

- In our example, we have to find **d** such that the above equation is satisfied. Which means, **5 d** = **1** mod **96** and also **d** < **96**.
- Solving for multiple values of **d** (can be calculated using the extended version of Euclid's algorithm), we can see that **d** = 77 satisfies our condition. See the math: $77 \times 5 = 385$ and $385 - 1 = 384$ is divisible by 96 because $4 \times 96 + 1 = 385$
- We can conclude that the in our example, the private key will be (77, 119).
- Now you have got your key pairs!

Encryption/Decryption Using Key Pair

Once the keys are generated, the process of encryption and decryption are fairly simple. The math behind them is as follows:

Encrypting the plaintext message **m** to get the ciphertext message **c** is as follows:

c = **m . e** (mod **n**) given the public key (**e**, **n**) and the plaintext message **m**.

Decrypting the ciphertext message **c** to get the plaintext message **m** is as follows:

m = **c . d** (mod **n**) given the private key (**d**, **n**) and the ciphertext **c**.

Note that RSA scheme is a block cipher where the input is divided into small blocks that the RSA algorithm can consume. Also, the plaintext and the ciphertext are all integers from **0** to **n − 1** for some integer **n** that is known to both sender and receiver. This means that the input plaintext is represented as integer, and when that goes through RSA and becomes ciphertext, they are again integers but not the same ones as input; we encrypted them remember? Now, considering the same key pairs from the

previous example, let us go through the steps to understand how it works practically:

- The sender wants to send a text message to the receiver whose public key is known and is say (**e**, **n**).
- The sender breaks the text message into blocks that can be represented as a series of numbers less than **n**.
- The ciphertext equivalents of plaintext can be found using **c** = **m e** (mod **n**). If the plaintext (**m**) is 19 and the public key is (5, 119) with **e** = 5 and **n** = 119, then the ciphertext **c** will be 195(mod 119) = 2, 476, 099 (mod 119) = 66, which is the remainder and 20,807 is the quotient, which we do not use. So, **c** = 66
- When the ciphertext 66 is received at the receiver's end, it needs to be decrypted to get the plaintext using **m** = **c d** (mod **n**).
- The receiver already has the private key (**d**, **n**) with **d** = 77 and **n** = 119, and received the ciphertext **c** = 66 by the sender. So, the receiver can easily retrieve the plaintext using these values as **m** = 6,677(mod 119) = 19
- For the modular arithmetic calculations, there are many online calculators that you can play around with, such as: http://comnuan.com/cmnn02/cmnn02008/

We looked at the math behind RSA algorithm. Now we know that **n** (supposed to be a very large number) is publicly available. Though it is public, factoring this large number to get the prime numbers **p** and **q** is extremely difficult. The RSA scheme is based on this practical difficulty of factoring large numbers. If **p** and **q** are not large enough, or the public key **e** is small, then the strength of RSA goes down. Currently, RSA keys are typically between 1024 and 2048 bits long. Note that the computational overhead of the RSA cryptography increases with the size of the keys.

In situations where the amount of data is huge, it is advisable to use a symmetric encryption technique and share the key using an asymmetric encryption technique such as RSA. Also, we looked at one of the aspects of RSA, that is, for encryption and decryption. However, it can also be used for authentication through digital signature. Just to give a high-level idea, one can take the hash of the data, sign it using their own private key, and share it along with the data. The receiver can check with the sender's public key and ensure that it was the sender who sent the data, and not someone else. This way, in addition to secure key transport, the public key encryption method RSA also offers authentication using a digital signature. Note here that a different algorithm called digital signature algorithm (DSA) can also be used in such situations that we will learn about in the following section.

RSA is widely being used with HTTPS on web browsers, emails, VPNs, and satellite TV. Also, many commercial applications or the apps in app stores are also digitally signed using RSA. SSH also uses public key cryptography; when you connect to an SSH server, it broadcasts a public key that can be used to encrypt data to be sent to that server. The server can then decrypt the data using its private key.

Digital Signature Algorithm

The DSA was designed by the NSA as part of the Digital Signature Standard (DSS) and standardized by the NIST. Note that its primary objective is to sign messages digitally, and not encryption. Just to paraphrase, RSA is for both key management and authentication whereas DSA is only for authentication. Also, unlike RSA, which is based on large-number factorization, DSA is based on discrete logarithms. At a high level, DSA is used as shown in Figure 2-15.

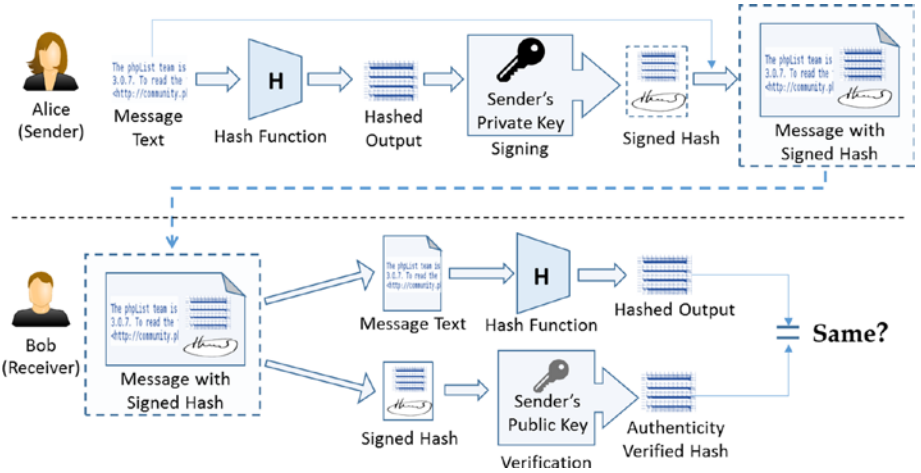


Figure 2-15. *Digital Signature Algorithm (DSA)*

As you can see in Figure 2-15, the message is first hashed and then signed because it is more secured compared with signing and then hashing it. Ideally, you would like to verify the authenticity before doing any other operation. So, after the message is signed, the signed hash is tagged with the message and sent to the receiver. The receiver can then check the authenticity and find the hash. Also, hash the message to get the hash again and check if the two hashes match. This way, DSA provides the following security properties:

- Authenticity: Signed by private key and verified by public key
- Data integrity: Hashes will not match if the data is altered.
- Non-repudiation: Since the sender signed it, they cannot deny later that they did not send the message. Non-repudiation is a property that is most desirable in situations where there are chances of a dispute over the exchange of data. For example, once an order is placed electronically, a purchaser cannot deny the purchase order if non-repudiation is enabled in such a situation.

A typical DSA scheme consists of three algorithms: (1) key generation, (3) signature generation, and (3) signature verification.

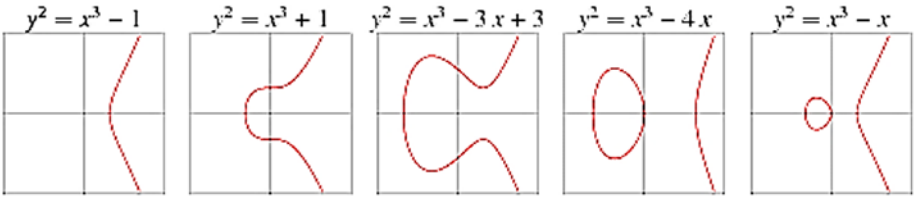
Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) actually evolved from Diffie-Hellman cryptography. It was discovered as an alternative mechanism for implementing public key cryptography. It actually refers to a suite of cryptographic protocols and is based on the discrete logarithm problem, as in DSA. However, it is believed that the discrete logarithmic problem is even harder when applied to the points on an elliptic curve. So, ECC offers greater security for a given key size. A 160-bit ECC key is considered to be as secured as a 1024-bit RSA key. Since smaller key sizes in ECC can provide greater security and performance compared with other public key algorithms, it is widely used in small embedded devices, sensors, and other IoT devices, etc. There are extremely efficient hardware implementations available for ECC.

ECC is based on a mathematically related set of numbers on an elliptic curve over finite fields. Also, it has nothing to do with ellipses! Mathematically, an elliptic curve satisfies the following mathematical equation:

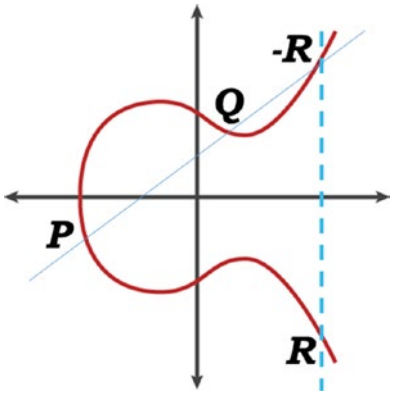
$y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$

With different values of "**a**" and "**b**", the curve takes different shapes as shown in the following diagram:

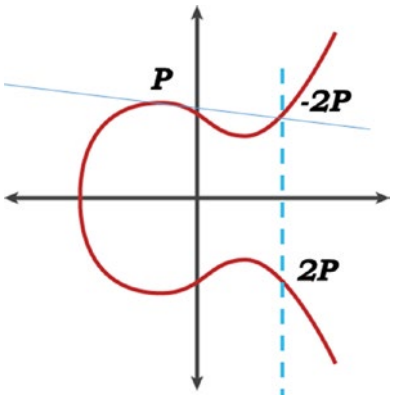


There are several important characteristics of elliptic curves that are used in cryptography, such as:

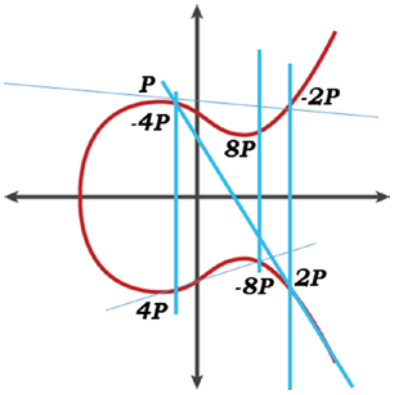
- They are horizontally symmetrical. i.e., what is below the X-axis is a mirror image of what is above the X-axis. So, any point on the curve when reflected over the X-axis still remains on the curve.
- Any nonvertical line can intersect the curve in at most three places.
- If you consider two points **P** and **Q** on the elliptic curve and draw a line through them, the line may exactly cross the curve at one more places. Let us call it (− **R**). If you draw a vertical line through (− **R**), it will cross the curve at, say, **R**, which is a reflection of the point (− **R**). Now, the third property implies that **P** + **Q** = **R**. This is called "point addition," which means adding two points on an elliptic curve will lead you to another point on the curve. Refer to the following diagram for a pictorial representation of these three properties.



- So, you can apply point addition to any two points on the curve. Now, in the previous bullet-point, we did point addition of **P** and **Q** (**P** + **Q**) and found − **R** and then ultimately arrived at **R**. Once we arrive at **R**, we can then draw a line from **P** to **R** and see that the line intersects the graph again at a third point. We can then take that point and move along a vertical line until it intersect the graph again. This becomes the point addition for points **P** and **R**. This process with a fixed **P** and the resulting point can continue as long as we want, and we will keep getting new points on the curve.
- Now, instead of two points **P** and **Q**, what if we apply the operation to the same point **P**, i.e., **P** and **P** (called "point doubling"). Obviously, infinite numbers of lines are possible through P, so we will only consider the tangential line. The tangent line will cross the curve in one more point and a vertical line from there will cross the curve again to get to the final value. It can be shown as follows:



- It is evident that we can apply point doubling "**n**" number of times to the initial point and every time it will lead us to a different point on the curve. The first time we applied point doubling to the point **P**, it took us to the resulting point **2P** as you can see in the diagram. Now, if the same is repeated "**n**" number of times, we will reach a point on the curve as shown in the following diagram:



- In the aforementioned scenario, when the initial and final point is given, there is no way one can say that the point doubling was applied "**n**" number of times to reach the final resulting point except trying for all possible "**n**" one by one. This is the discrete logarithm problem for ECC, where it states that given a point **G** and **Q**, where **Q** is a multiple of **G**, find "**d**" such that **Q** = **d G**. This forms the one-way function with no shortcuts. Here, **Q** is the public key and **d** is the private key. Can you extract private key **d** from public key **Q**? This is the elliptic curve discrete logarithm problem, which is computationally difficult to solve.

- Further to this, the curve should be defined over a finite field and not take us to infinity! This means the "max" value on the X-axis has to be limited to some value, so just roll the values over when we hit the maximum. This value is represented as **P** (not the **P** used in the graphs here) in the ECC cryptosystem and is called "modulo" value, and it also defines the key size, hence the finite field. In many implementations of ECC, a prime number for "**P**" is chosen.
- Increased size of "**P**" results in more usable values on the curve, hence more security.
- We observed that point addition and point doubling form the basis for finding the values that are used for encryption and decryption.

So, in order to define an ECC, the following domain parameters need to be defined:

- The Curve Equation: $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$
- **P**: The prime number, which specifies the finite field that the curve will be defined over (modulo value)
- **a** and **b**: Coefficients that define the elliptic curve
- **G**: Base point or the generator point on the curve. This is the point where all the point operations begin and it defines the cyclic subgroup.
- **n**: The number of point operations on the curve until the resultant line is vertical. So, it is the order of **G**, i.e., the smallest positive number such that $\mathbf{nG} = \infty$. It is normally prime.
- **h**: It is called "cofactor," which is equal to the order of the curve divided by **n**. It is an integer value and usually close to 1.

Note that ECC is a great technique to generate the keys, but is used alongside other techniques for digital signatures and key exchange. For example, Elliptic Curve Diffie-Hellman (ECDH) is quite popularly used for key exchange and ECDSA is used for digital signatures.

Elliptic Curve Digital Signature Algorithm

The ECDSA is a type of DSA that uses ECC for key generation. As the name suggests, its purpose is digital signature, and not encryption. ECDSA can be a better alternative to RSA in terms of smaller key size, better security, and higher performance. It is one of the most important cryptographic components used in Bitcoins!

We already looked at how digital signatures are used to establish trust between the sender and receiver. Since authenticity of the sender and integrity of the message can be verified through digital signatures, two unknown parties can transact with each other. Note that the sender and the receiver have to agree on the domain parameters before engaging in the communication.

There are broadly three steps to ECDSA: key generation, signature generation, and signature verification.

Key Generation

Since the domain parameters (**P**, **a**, **b**, **G**, **n**, **h**) are preestablished, the curve and the base point are known by both parties. Also, the prime **P** that makes it a finite field is also known (**P** is usually 160 bits and can be greater as well). So, the sender, say, Alice does the following to generate the keys:

- Select a random integer **d** in the interval [**1**, **n** − **1**]
- Compute **Q** = **d G**
- Declare **Q** is the public key and keep **d** as the private key.

Signature Generation

Once the keys are generated, Alice, the sender, would use the private key "**d**" to sign the message (**m**). So, she would perform the following steps in the order specified to generate the signature:

- Select a random number **k** in the interval [**1**, **n − 1**]
- Compute **k.G** and find the new coordinates (x_1, y_1) and find **r** = x_1 mod **n**

  If **r** = 0, then start all over again
- Compute **e** = SHA-1 (**m**)
- Compute **s** = $\mathbf{k}^{-1}$ (**e** + **d . r**) mod **n**

  If s = 0, then start all over again from the first step
- Alice's signature for the message (**m**) would now be (**r**, **s**)

Signature Verification

Let us say Bob is the receiver here and has access to the domain parameters and the public key **Q** of the sender Alice. As a security measure, Bob should first verify that the data he has, which is the domain parameters, the signature, and Alice's public key **Q** are all valid. To verify Alice's signature on the message (**m**), Bob would perform the following operations in the order specified:

- Verify that **r** and **s** are integers in the interval [**1**, **n − 1**]
- Compute **e** = SHA-1 (**m**)
- Compute **w** = $\mathbf{s}^{-1}$ mod **n**
- Compute $\mathbf{u_1}$ = **e w** mod **n,** and $\mathbf{u_2}$ = **r w** mod **n**
- Compute **X** = $\mathbf{u_1}$ G + $\mathbf{u_2}$ G, where X represents the coordinates, say (x_2, y_2)

- Compute $\mathbf{v} = x_1 \bmod \mathbf{n}$
- Accept the signature if $\mathbf{r} = \mathbf{v}$, otherwise reject it

In this section, we looked at the math behind ECDSA. Recollect that we used a random number while generating the key and the signature. It is extremely important to ensure that the random numbers generated are actually cryptographically random. In many use cases, 160-bit ECDSA is used because it has to match with the SHA-1 hash function.

Out of so many use cases, ECDSA is used in digital certificates. In its simplest form, a digital certificate is a public key, bundled with the device ID and the certificate expiration date. This way, certificates enable us to check and confirm to whom the public key belongs and the device is a legitimate member of the network under consideration. These certificates are very important to prevent "impersonation attack" in key establishment protocols. Many TLS certificates are based on ECDSA key pair and this usage continues to grow.

Code Examples of Assymetric Key Cryptography

Following are some code examples of different public ley algorithms. This section is just intended to give you a heads-up on how to use different algorithms programatically. Code examples are in Python but would be quite similar in different languages; you just have to find the right library functions to use.

```
# -*- coding: utf-8 -*-
import Crypto
from Crypto.PublicKey import RSA
from Crypto import Random
from hashlib import sha256
```

```
# Function to generate keys with default lenght 1024
def generate_key(KEY_LENGTH=1024):
    random_value= Random.new().read
    keyPair=RSA.generate(KEY_LENGTH,random_value)
    return keyPair

#Generate Key for ALICE and BOB
bobKey=generate_key()
aliceKey=generate_key()

#Print Public Key of Alice and Bob. This key could shared
alicePK=aliceKey.publickey()
bobPK=bobKey.publickey()

print "Alice's Public Key:", alicePK
print "Bob's Public Key:", bobPK

#Alice wants to send a secret message to Bob. Lets create a
dummy message for Alice
secret_message="Alice's secret message to Bob"
print "Message",  secret_message

# Function to generate a signature
def generate_signature(key,message):
    message_hash=sha256(message).digest()
    signature=key.sign(message_hash,'')
    return signature

# Lets generate a signature for secret message
alice_sign=generate_signature(aliceKey,secret_message)

# Before sending message in network, encrypt message using the
Bob's public key...
encrypted_for_bob  = bobPK.encrypt(secret_message, 32)
```

```
# Bob decrypts secret message using his own private key...
decrypted_message   = bobKey.decrypt(encrypted_for_bob)
print "Decrypted message:", decrypted_message

# Bob will use the following function to verify the signature
from Alice using her public key
def verify_signature(message,PublicKey,signature):
    message_hash=sha256(message).digest()
    verify = PublicKey.verify(message_hash,signature)
    return verify

# bob is verifying using decrypted message and alice's public
key
print "Is alice's signature for decrypted message valid?",
verify_signature(decrypted_message,alicePK, alice_sign)
```

The ECDSA Algorithm

```
import ecdsa

# SECP256k1 is the Bitcoin elliptic curve
signingKey = ecdsa.SigningKey.generate(curve=ecdsa.SECP256k1)
# Get the verifying key
verifyingKey = signingKey.get_verifying_key()

# Generate The signature of a message
signature = signingKey.sign(b"signed message")

# Verify the signature is valid or invalid for a message
verifyingKey.verify(signature, b"signed message") # True.
Signature is valid

# Verify the signature is valid or invalid for a message
assert verifyingKey.verify(signature, b"message") # Throws an
error. Signature is invalid for message
```

Diffie-Hellman Key Exchange

We already looked at symmetric key cryptography in the previous sections. Recollect that sharing the secret between the sender and the receiver is a very big challenge. As a rule of thumb, we are now aware that the the communication channel is always insecure. There could always be an *Eve* trying to intercept your message while it is being transmitted by using various different kinds of attacks. So, the technique of DH was developed for securely exchanging the cryptographic keys. Obviously, you must be wondering how secure key exchange is possible when the communication channel itself is insecured. Well, later in this section you will see that the DH technique is not really sharing the entire secret key between two parties, rather it is about creating the key together. At the end of the day, what is important is that the sender and the receiver both have the same key. However, keep in mind that it is not asymmetric key cryptography, as encryption/decrytion does not take place during the exchange. In fact, it was the base upon which asymmetric key cryptography was later designed. The reason we are looking at this technique now is because a lot of math that we already studied in the previous section is useful here.

Let us first try to understand the concept at a high level before getting into the mathematical explanation. Take a look at the following (Figure 2-16), where a simple explanation of DH algorithm is presented with colors.

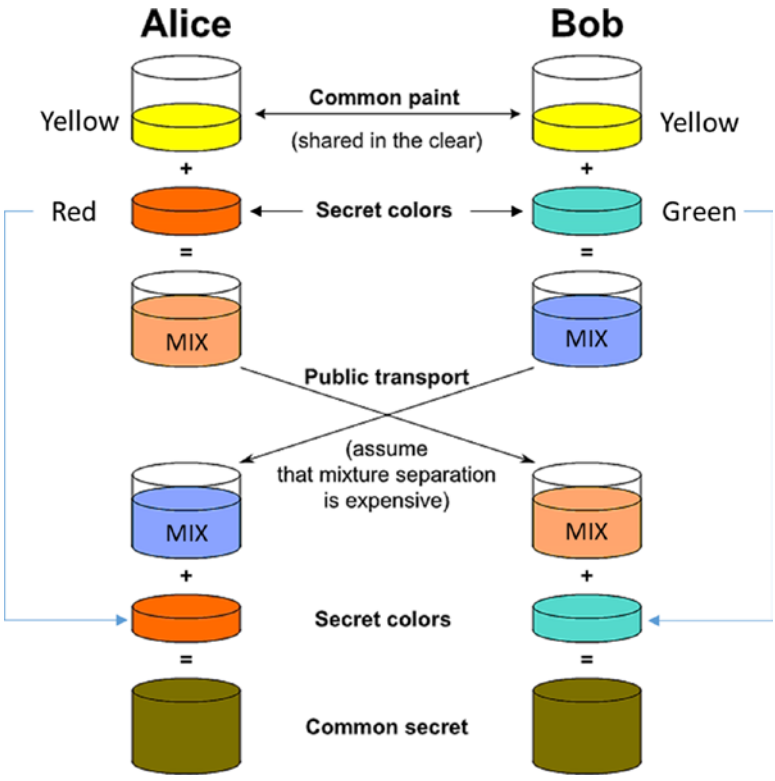


Figure 2-16. *Diffie-Hellman key exchange illustration*

Notice that only the yellow color was shared between the two parties in the first step, which may represent any other color or a randon number. Both parties then add their own secret to it and make a mixture. That mixture is again shared through the same insecured channel. Respective parties then add their secret to it and form their final common secret. In this example with colors, observe that the common secrets are the combination of same sets of colors. Let us now look at the actual mathematical steps that take place for the generation of keys:

- Alice and Bob agree on $P = 23$ and $G = 9$
- Alice chooses private key $a = 4$, computes $9^4 \bmod 23 = 6$ and sends it to Bob

- Bob chooses private key b = 3, computes $9^3 \bmod 23 = 16$ and sends it to Alice
- Alice computes $16^4 \bmod 23 = 9$
- Bob computes $6^3 \bmod 23 = 9$

If you follow through these steps, you will find that both Alice and Bob are able to generate the same secret key at their ends that can be used for encryption/decryption. We used small numbers in this example for easy understanding, but large prime numbers are used in real-world use cases. To understand it better, let us go through the following code snippet and see how DH algorithm can be implemented in a simple way:

```
/* Program to calculate the Keys for two parties using Diffie-
Hellman Key exchange algorithm */

// function to return value of a ^ b mod P
long long int power(long long int a, long long int b, long long
int P)
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

//Main program for DH Key computation
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Both the parties agree upon the public keys G and P
    P = 23; // A prime number P is taken
    printf("The value of P : %lld\n", P);
```

```
    G = 9; // A primitve root for P, G is taken
    printf("The value of G : %lld\n\n", G);

    // Alice will choose the private key a
    a = 4; // a is the chosen private key
    printf("The private key a for Alice : %lld\n", a);
    x = power(G, a, P); // gets the generated key

    // Bob will choose the private key b
    b = 3; // b is the chosen private key
    printf("The private key b for Bob : %lld\n\n", b);
    y = power(G, b, P); // gets the generated key

    // Generating the secret key after the exchange of keys
    ka = power(y, a, P); // Secret key for Alice
    kb = power(x, b, P); // Secret key for Bob


    printf("Secret key for the Alice is : %lld\n", ka);
    printf("Secret Key for the Bob is : %lld\n", kb);

    return 0;
}
```

Note While the discrete logarithm problem is traditionally used (the x^y mod p), the general process can be modified to use elliptic curve cryptography as well.

Symmetric vs. Asymmetric Key Cryptography

We looked at various aspects and types of both symmetric and asymmetric key algorithms. Obviously, their design goals and implications are different. Let us have a comparative analysis so that we use the right one at the right place.

- Symmetric key cryptography is also referred to as private key cryptography. Similarly, asymmetric key cryptography is also called public key cryptography.
- Key exchange or distribution in symmetric key cryptography is a big headache, unlike asymmetric key cryptography.
- Asymmetric encryption is quite compute-intensive because the length of the keys is usually large. Hence, the process of encryption and decryption is slower. On the contrary, symmetric encryption is faster.
- Symmetric key cryptography is appropriate for long messages because the speed of encryption/decryption is fast. Asymmetric key cryptography is appropriate for short messages, and the speed of encryption/decryption is slow.
- In symmetric key cryptography, symbols in plaintext and ciphertext are permuted or substituted. In asymmetric key cryptography, plaintext and ciphertext are treated as integers.
- In many situations, when symmetric key is used for encryption and decryption, asymmetric key technique is used to share and agree upon the key used in encryption.

- Asymmetric key cryptography finds its strongest application in untrusted environments, when parties involved have no prior relationship. Since the unknown parties do not get any prior opportunity to establish shared secret keys with each other, sharing of sensitive data is secured through public key cryptography.
- Symmetric cryptographic techniques do not provide a way for digital signatures, which are only possible through asymmetric cryptography.
- Another good case is the number of keys required among a group of nodes to communicate with each other. How many keys do you think would be needed among, say, 100 participants when symmetric key cryptography is needed? This problem of finding the keys needed can be approached as a complete graph problem with order 100. Like each vertex requires 99 connected edges to connect with everyone, every participant would need 99 keys to establish secured connections with all other nodes.

  So, in total, the keys needed would be $100 * (100 - 1)/2 = 4{,}950$. It can be generalized for "**n**" number of participants as $\mathbf{n} * (\mathbf{n} - 1)/2$ keys in total. With an increased number of participant, it becomes a nightmare! However, in the case of asymmetric key cryptography, each participant would just need two keys (one private and one public). For a network of 100 participants, total keys needed would be just 200. Table 2-5 shows some sample data to give you an analogy on the increased number of keys needed when the number of participants increases.

Table 2-5. *Key Requirements Comparison for Symmetric and Asymmetric Key Techniques*

Number of Participants	Number of Symmetric Keys	Number of Asymmetric Keys
2	1	4
4	6	8
10	45	20
50	1225	100
100	4950	200
1000	499500	2000

Game Theory

Game Theory is a certainly quite an old concept and is being used in many real-life situations to solve complex problems. The reason we are covering this topic at a high level is because it is used in Bitcoins and many other blockchain solutions. It was formally introduced by John von Neumann to study economic decisions. Later, it was more popularized by John Forbes Nash Jr because of his theory of "Nash Equilibrium," which we will look into shortly. Let us first understand what game theory is.

Game theory is a theory on games, where the games are not just what children play. Most are situations where two or more parties are involved with some strategic behavior. Examples: A cricket tournament is a game, two conflicting parties in a court of law with lawyers and juries is a game, two siblings fighting over an ice cream is a game, a political election is a game, a traffic signal is also a game. Another example: Say you applied for a blockchain job and you are selected and offered a job offer with some salary, but you reject the offer, thinking there is a huge gap in the demand and supply and chances are good they will revise the offer with a higher salary. You must be thinking now, what is not a game? Well, in real situations, almost everything is a game. So, a "game" can be defined as a situation involving a "correlated rational choice." What it means is that the

prospects available for any player are dependent not only on their own choices, but also on the choices that others make in a given situation. In other words, if your fate is impacted by the actions of others, then you are in a game. So what is game theory?

Game theory is a study of strategies involved in complex games. It is the art of making the best move, or opting for a best strategy in a given situation based on the objective. To do so, one must understand the strategy of the opponent and also what the opponent thinks your move is going to be. Let us take a simple example: There are two siblings, one elder and the other younger. Now, there are two ice creams in the fridge, one is orange flavor and the other is mango flavor. The elder one wants to eat the orange flavor, but knows if he opts for that, then the younger one would cry for the same orange. So, he opts for the mango flavored ice cream and it turns out as expected, the younger one wants the same. Now, the elder one pretends to have sacrificed the mango flavored ice cream and gives it to the younger one and eats the orange one himself. Look at the situation: this is a win-win for both the parties, as this was the objective of the elder one. If the elder one wanted, he could simply have fought with the younger kid and got the orange one if that was his objective. In the second case, the elder one would strategize where to hit so that the younger kid is not injured much but enough so that he gives up on the orange flavored ice cream. This is game theory: what is your objective and what should be your best move?

One more example: more on a business side this time. Imagine that you are a vendor supplying vegetables to a town. There are, say, three ways to get to the town, out of which one is a regular route in the sense that everyone goes by that route, maybe because it is shorter and better. One day, you see that the regular route has been blocked because of some repair activity and in no way can you go by that route. You are now left with two other routes. One of those is a short route to the destination town but is a little narrow. The other one is a little longer route but wide enough. Here, you have to make a strategy as to which route of the two you

need to go by. The situation may be such that there is heavy traffic on the roads and many people would try to get through the shortest route. This can lead to heavy congestion on that route and can cause a huge delay. So, you decided to take the longer route to reach the town on time, but at the cost of few extra dollars spent on fuel. You are sure you can easily get compensated for that if you arrive on time and sell your vegetables early at a good price. This is game theory: what is your best move for the objective you have in mind, which is usually finding an optimal solution.

In many situations, the role that you play and your objective both play a vital role in formulating the strategy. Example: If you are an organizer of a sport event, and not a participant in the competition, then you would formulate a strategy where your objective could be that you want the participants to play by the rules and follow the protocol. This is because you do not care who wins at the end, you are just an organizer. On the other hand, a participant would strategize the winning moves by taking into account the strengths and weaknesses of the opponent, and the rules imposed by the organizer because there could be penalties if you break the rules. Now, let us consider this situation with you playing the role of the organizer. You should consider if there could be a situation where a participant breaks a rule and loses one point but injures the opponent so much that they cannot compete any longer. So, you have to take into account what the participants can think and set your rules accordingly.

Let us try to define game theory once again based on what we learned from the previous examples. It is the method of modeling real-life situations in the form of a game and analyzing what the best strategy or move of a person or an entity could be in a given situation for a desired outcome. Concepts from game theory are widely used in almost every aspect of life, such as politics, social media, city planning, bidding, betting, marketing, distributed storage, distributed computing, supply chains, and finance, just to name a few. Using game theoretic concepts, it is possible to design systems where the participants play by the rules without assuming emotional or moral values of them. If you want to go beyond just building

a proof of concept and get your product or solution to production, then you should prioritize game theory as one of the most important elements. It can help you build robust solutions and lets you test those with different interesting scenarios. Well, many people already think in game theoretic perspectives without knowing it is game theory. However, if you are equipped with the many tools and techniques from game theory, it definitely helps.

Nash Equilibrium

In the previous section, we looked at different examples of games. There are many ways to classify games, such as cooperative/noncooperative games, symmetric/asymmetric games, zero-sum/non-zero-sum games, simultaneous/sequential games, etc. More generally, let us focus on the cooperative/noncooperative perspective here, because it is related to the Nash equilibrium.

As the name suggests, the players cooperate with each other and can work together to form an alliance in cooperative games. Also, there can be some external force applied to ensure cooperative behavior among the players. On the other hand, in noncooperative games, the players compete as individuals with no scope to form an alliance. The participants just look after their own interests. Also, no external force is available to enforce cooperative behavior.

Nash equilibrium states that, in any noncooperative games where the players know the strategies of each other, there exists at least one equilibrium where all the players play their best strategies to get the maximum profits and no side would benefit by changing their strategies. If you know the strategies of other players and you have your own strategy as well, if you cannot benefit by changing your own strategy, then this is the state of Nash equilibrium. Thus, each strategy in a Nash equilibrium is a best response to all other strategies in that equilibrium.

Note that a player may strategize to win as an individual player, but not to defeat the opponent by ensuring the worst for the opponents. Also, any game when played repeatedly may eventually fall into the Nash equilibrium.

In the following section, we will look at the "prisoner"s dilemma" to get a concrete understanding of the Nash equilibrium.

Prisoner's Dilemma

Many games in real life can also be non-zero-sum games. Prisoner's dilemma is one such example, which can be broadly categorized as a symmetric game. This is because, if you change the identities of the players (e.g., if two players "A" and "B" are playing, then "A" becomes "B" and "B" becomes "A"), and also the strategies do not change, then the payoff remains the same. This is what a symmetric game is.

Let us start directly with an example. Assume that there are two guys, Bob and Charlie, who are caught by the cops for selling drugs independently, say in different locations. They are kept in two different cells for interrogation. They were then toldd that they would be sentenced to jail for two years for this crime. Now, the cops somehow suspect that these two guys could also be involved in the robbery that just happened last week. If they did not do the robbery, then it is two years of imprisonment anyway. So, the cops have to strategize a way to get to the truth. So here is what they do.

The cops go to Bob and give him a choice, a good choice that goes like this. If Bob confesses his crime and Charlie does not, then his punishment would go down from two years to just one year and Bob gets five years. However, if Bob denies and Charlie confesses, then Bob gets five years and Charlie gets just one year. Also, if both confess, then both get three years of imprisonment. Similarly, the same choice is given to Charlie as well. What do you think they are going to do? This situation is called the prisoner's dilemma.

Both Bob and Charlie are in two different cells. They cannot talk to each other and conclude with the situation where they both deny and get two years in jail (just for the drug dealing case), which seems to be the global optimum in this situation. Well, even if they could talk to each other, they may not really trust each other.

What would go through Bob's mind now? He has two choices, confess or deny. He knows that Charlie would choose what is best for him, and he himself is no different. If he denies and Charlie confesses, then he is in trouble by getting five years of jail and Charlie gets just one year of jail. He certainly does not want to get into this situation.

If Bob confesses, then Charlie has two choices: confess or deny. Now Bob thinks that if he confesses, then whatever Charlie does, he is not getting more than three years. Let us state these scenarios for Bob.

- Bob confesses and Charlie denies—Bob gets one year, Charlie gets five years (best case given Bob confesses)
- Bob confesses and Charlie also confesses—Both Bob and Charlie get three years (worst case given Bob confesses)

This situation is called Nash equilibrium where each party has taken the best move, given the choices of the other party. This is definitely not the global optimum, but represents the best move as an individual. Now, if you look at this situation as an outsider, you would say both should deny and get two years. But when you play as a participant in the game, Nash equilibrium is what you would eventually fall into. Note that this is the most stable stage where you changing your decision does not benefit you at all. It can be pictorially represented as shown in Figure 2-17.

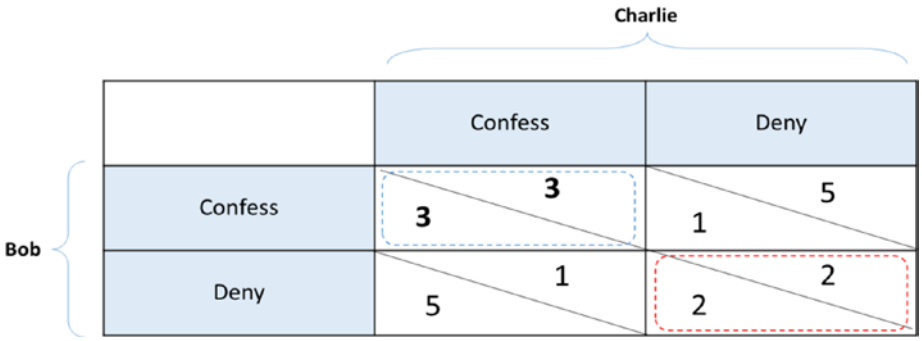


Figure 2-17. *Prisoner's dilemma–payoff matrix*

Byzantine Generals' Problem

In the previous section, we looked at different examples of games and learned a few game theory concepts. Now we will discuss a specific problem from the olden days that is still widely used to solve many computer science as well as real-life problems.

The Byzantine Generals' Problem was a problem faced by the Byzantine army while attacking a city. The situation was straightforward yet very difficult to deal with. To put it simply, the situation was that several army factions commanded by separate generals surrounded a city to win over it. The only chance of victory is when all the generals attack the city together. However, the problem is how to reach a consensus. This implies that either all the generals should attack or all of them should retreat. If some of them attack and some retreat, then chances are greater they would lose the battle. Let us take an example with numbers to be able to understand the situation better.

Let us assume a situation where there are five factions of the Byzantine army surrounding a city. They would attack the city if at least three out of five generals are willing to attack, but retreat otherwise. If there is a traitor among the generals, what he can do is vote for attack with the generals willing to attack and vote for retreat with the generals willing to retreat. He can do so because the army is dispersed in factions, which makes

centralized coordination difficult. This can result in two generals attacking the city and getting outnumbered and defeated. There could be more complicated issues with such a situation:

- What if there is more than one traitor?
- How would the message coordination between generals take place?
- What if a messenger is caught/killed/bribed by the city commander?
- What if a traitor general forges a different message and fools other generals?
- How to find the generals who are honest and who are traitors?

As you can see, there are so many challenges that need to be addressed for a coordinated attack on the city. It can be pictorially represented as in Figure 2-18.

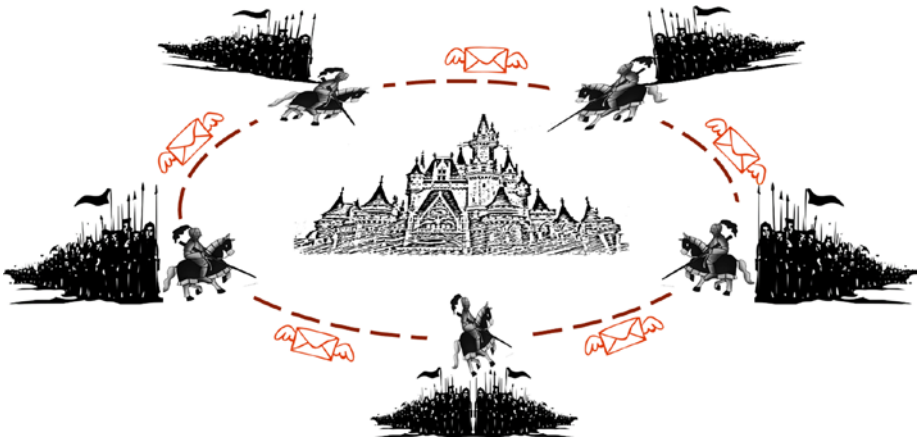


Figure 2-18. *Byzantine army attacking the city*

There are numerous scenarios in real life that are analogous to the Byzantine Generals' Problem. How a group of people reach consensus on some voting agenda or how to maintain the consistent state of a distributed or decentralized database, or maintaining the consistent state of blockchain copies across nodes in a network are a few examples similar to the Byzantine Generals' Problem. Note, however, that the solutions to these different problems could be quite different in different situations. We will look at how Bitcoin solves the Byzantine Generals' Problem later in this book.

Zero-Sum Games

A zero-sum game in game theory is quite straightforward. In such games, one player's gain is equivalent to another player's loss. Example: One wins exactly the same amount as the opponent loses, which means choices by players can neither increase nor decrease the available resources in a given situation.

Poker, Chess, Go, etc. are a few examples of zero-sum games. To generalize even more, the games where only one person wins and the opponent loses, such as tennis, badminton, etc. are also zero-sum games. Many financial instruments such as swaps, forwards, and options can also be described as zero-sum instruments.

In many real-life situations, gains and losses are difficult to quantify. So, zero-sum games are less common compared with non-zero-sum games. Most financial transactions or trades and the stock market are non-zero-sum games. Insurance, however, is a field where a zero-sum game plays an important role. Just think about how the insurance schemes might work. We pay an insurance premium to the insurance companies to guard against some difficult situations such as accidents, hospitalization, death, etc. Thinking that we are insured, we live a peaceful life and we are fairly compensated by the insurance companies when we face such tough situations. There is certainly a financial backup that helps us survive.

Note that everyone who pays the premium does not meet with accident or get hospitalized, and the ones who do need a lot of money compared with the premium they pay. You see, things are quite balanced here, even considering the operational expenses of the insurance company. Again, the insurance company may invest the premium we pay and get some return on that. Still, this is a zero-sum game.

Just to give you a different example, if there is one open position for which an interview drive is happening, then the candidate who qualifies actually does it at the cost of others' disqualification. This is also a zero-sum game.

You may ask if there is any use in studying about zero-sum games. Just being aware of a zero-sum situation is quite useful in understanding and devising a strategy for any complex problem. We can analyze if we can practically gain in a given situation in which the transactions are taking place.

Why to Study Game Theory

Game theory is a revolutionary interdisciplinary phenomenon bringing together psychology, economics, mathematics, philosophy, and an extensive mix of various other academic areas.

We say that game theory is related to real-world problems. However, the problems are limitless. Are the game theoretic concepts limitless as well? Certainly! We use game theory every day, knowingly or unknowingly, because we always use our brains to take the best strategic action, given a situation. Don't we? If that is so, why study game theory?

Well, there are numerous examples in game theory that help us think differently. There are some theories developed such as Nash Equilibrium that relate to many real-life situations. In many real-world situations, the participants or the players are faced with a decision matrix similar to that of a "prisoner's dilemma." So, learning these concepts not only helps us formulate the problems in a more mathematical way, but also enables

us to make the best move. It lets us identify aspects that each participant should consider before choosing a strategic action in any given interaction. It tells us to identify the type of game first; who are the players, what are their objectives or goals, what could be their actions, etc., to be able to take the best action. Much decision-making in real life involves different parties; game theory provides the basis for rational decision-making.

The Byzantine Generals' Problem that we studied in the previou section is widely used in distributed storage solutions and data centers to maintain data consistency across computing nodes.

Computer Science Engineering

As mentioned already, it is clever engineering with the concepts from computer science that stitches the components of cryptography, game theory, and many others to build a blockchain. In this section, we will learn some of the important computer science components that are used in blockchain.

The Blockchain

As we will see, a blockchain is actually a blockchain data structure; in the sense that it is a chain of blocks linked together. When we say a block, it can mean just a single transaction or multiple transactions clubbed together. We will start our discussion with hash pointers, which is the basic building block of blockchain data structure.

A hash pointer is a cryptographic hash pointing to a data block, where the hash pointer is the hash of the data block itself (Figure 2-19). Unlike linked lists that point to the next block so you can get to it, hash pointers point to the previous data block and provide a way to verify that the data has not been tampered with.

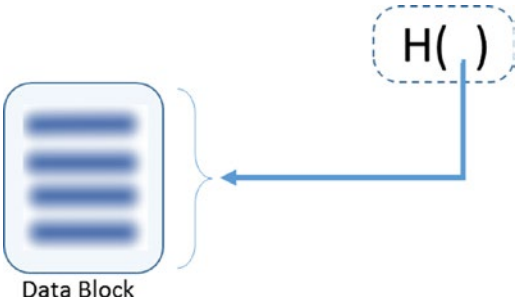


Figure 2-19. *Hash pointer for a block of transactions*

The purpose of the hash pointer is to build a tamper resistant blockchain that can be considered as a single source of truth. How does blockchain achieve this objective? The way it works is that the hash of the previous block is stored in the current block header, and the hash of the current block with its block header will be stored in the next block's header. This creates the blockchain as we can see in Figure 2-20.

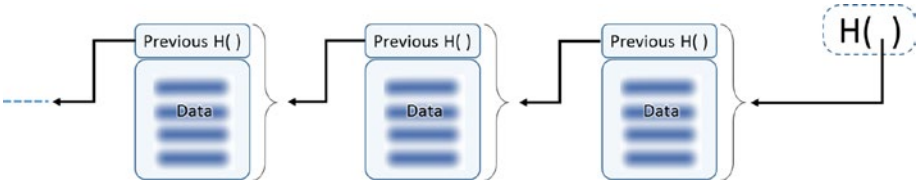


Figure 2-20. *Blocks in a blockchain linked through hash pointers*

As we can observe, every block points to its previous block, known as "the parent block." Every new block that gets added to the chain becomes the parent block for the next block to be added. It goes all the way to the first block that gets created in the blockchain, which is called "the genesis block." In such a design where blocks are linked back with hashes, it is practically infeasible for someone to alter data in any block. We already looked at the properties of hash functions, so we understand that the hashes will not match if the data is altered. What if someone changes the hash as well? Let us focus on Figure 2-21 to understand how it is not possible to alter the data in any way.

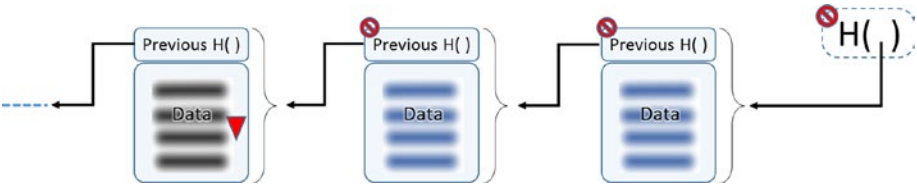


Figure 2-21. *Any attempt in changing Header or Block content breaks the entire chain. Assume that you altered the data in block-1234. If you do so, the hash that is stored in the block header of block-1235 would not match.*

- What if you also change the hash stored in the block header of block-1235 so that it perfectly matches the altered data. In other words, you hash the data block-1234 after you alter it and replace that new hash with the one stored in block header of block-1235. After you do this, the hash of the block-1235 changes (because block-1235 means the data and the header together) and it does not match with the one stored in the block header of block-1236.
- One has to keep doing this all the way till the final or the most recent hash. Since everyone or many in the network already have a copy of the blockchain along with the most recent hash, in no way is it possible to hack into the majority of the systems and change all the hashes at a time.
- This makes it a tamper-proof blockchain data structure.

This clearly means that each block can be uniquely identified by its hash. To calculate this hash, you can use either the SHA2 or SHA3 family of hash functions that we discussed in the cryptography section. If you use SHA-256 to hash the blocks, it would produce a 256-bit hash output such as:

```
000000000000000a73b6a2af7bad40ec3fc2a83dafd76ef15f3d1b71a7132765
```

Notice that there are only 64 characters in it. Since the hashed output is represented using hexadecimal characters, and every hex digit can be represented using four bits, the output is 64 × 4 = 256 bits. You would usually see that the 256-bit hashed output is represented using the 64 hex characters in many places.

The structure of a block, that is, block size, the data and header sections, number of transactions in a block, etc., is something that you should decide while designing a blockchain solution. For existing blockchains such as Bitcoin, Ethereum, or Hyperledger, the structure is already defined and you have to understand that to build on top of these platforms. We will take a closer look at the Bitcoin and Ethereum blockchains later in this book.

Merkle Trees

A Merkle tree is a binary tree of cryptographic hash pointers, hence it is a binary hash tree. It is named so after its inventor Ralph Merkle. It is another useful data structure being used in blockchain solutions such as Bitcoin. Merkle trees are constructed by hashing paired data (usually transactions at the leaf level), then again hashing the hashed outputs all the way up to the root node, called the Merkle root. Like any other tree, it is constructed bottom-up. In Bitcoin, the leaves are always transactions of a single block in a blockchain. We will discuss in a little while the advantages of using Merkle trees, so you can decide for yourself if the leaves would be transactions or a group of transactions in blocks. A typical Merkle tree can be represented as in Figure 2-22.

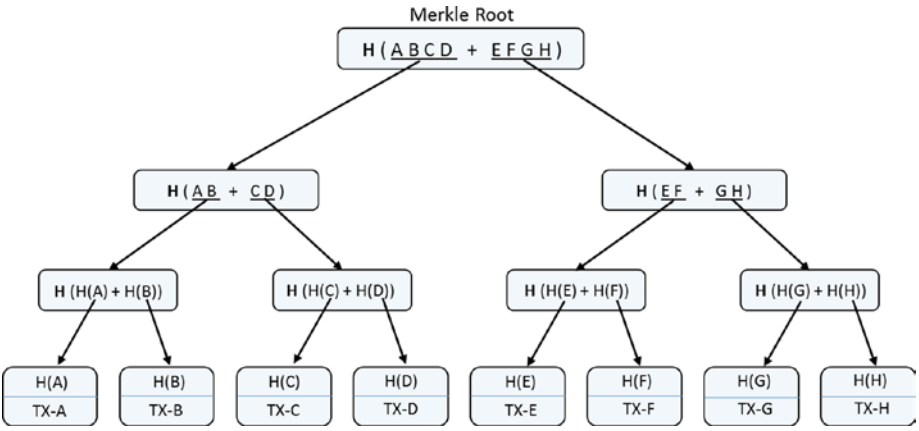


Figure 2-22. *Merkle tree representation*

Similar to the hash pointer data structure, the Merkle tree is also tamper-proof. Tampering at any level in the tree would not match with the hash stored at one level up in the hierarchy, and also till the root node. It is really difficult for an adversary to change all the hashes in the entire tree. It also ensures the integrity of the order of transactions. If you change just the order of the transactions, then also the hashes in the tree till the Merkle root will change.

Here is a situation. The Merkle tree is a binary tree and there should be an even number of items at the leaf level. What if there are an odd number of items? One good solution would be to duplicate the last transaction hash. Since it is the hash we are duplicating, it would mean just the same transaction and not create any issue such as double-spend or repeated transactions. That way, it is possible to balance the tree.

In the blockchain we discussed, if we were to find a transaction through its hash, or check if a transaction had happened in the past, how would we get to that transaction? The only way is to keep traversing till you encounter the exact block that matches the hash of the transaction. This is a case where a Merkle tree can help a great deal.

Merkle trees provide a very efficient way to verify if a specific transaction belongs to a particular block. If there are "n" transactions in a Merkle tree (leaf items), then this verification takes just Log (n) time as shown in Figure 2-23.

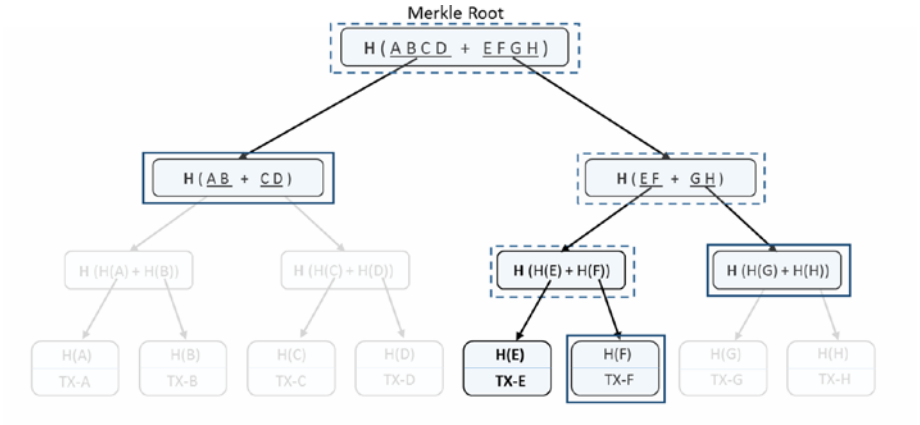


Figure 2-23. *Verification in Merkle tree*

To verify if a transaction or any other leaf item belongs to a Merkle tree, we do not need all items and the whole tree. Rather, a subset of it is needed as we can see in the diagram in Figure 2-23. One can just start with the transaction to verify along with its sibling (it is a binary tree so there would be one sibling leaf item), calculate the hash of those two, and see if it matches their parent hash. Then continue with that parent hash and its sibling at that level and hash them together to get their parent hash. Continuing this process all the way to the top root hash is the quickest possible way for transaction verification (just Log (n) time for n items). In the figure, only the solid rectangles are required and the dotted rectangles can be just computed, provided the solid rectangle data. Since there are eight transaction elements (n = 8), only three computations (log2 8 = 3) would be required for verification.

Now, how about a hybrid of both blockchain data structure and Merkle tree? Imagine a situation in a blockchain where each block has a lot of transactions. Since it is a blockchain, the hash of the previous block is already there; now, including the Merkle root of all the transactions in a block can help in quicker verification of the transactions. If we have to verify a transaction that is claimed to be from, say, block-22456, we can get the transactions of that block, verify the Merkle tree, and confirm quickly if that transaction is valid. We already saw that verifying a transaction is quite easy and fast with Merkle trees. Though blocks in the blockchain are tamper resistant and do not provide even the slightest scope to change anything in a block, the Merkle tree also ensures that the order of transactions is preserved.

In a typical blockchain setting, there could be many situations where a node (for simplicity sake, assume any node that does not have the full blockchain data, i.e., a light node) has to verify if a certain transaction took place in the past. There are actually two things that need verification here: transaction as part of the block, and block as part of the blockchain. To do so, a node does not have to download all the transactions of a block, it can simply ask the network for the information pertaining to the hash of the block and the hash of the transaction. The peers in the network who have the relevant information can respond with the Merkle path to that transaction. Well, you might ask how to trust the data that an unknown peer in the network is sharing with you. You already know that the hash functions are one-way. So in no way can an adversarial node forge transactions that would match a given hash value; it is even difficult to do so from transaction level till the Merkle root.

The use of Merkle trees is not limited to just blockchains: they are widely used in many other applications such as BitTorrent, Cassandra—an NoSQL database, Apache Wave, etc.

Example Code Snippet for Merkletree

This section is just intended to give you a heads-up on how to code up a Merkle tree at its most basic level. Code examples are in Python but would be quite similar in different languages; you just have to find the right library functions to use.

```
# -*- coding: utf-8 -*-

from hashlib import sha256

class MerkelTree(object):
    def __init__(self):
        pass

    def chunks(self,transaction,n):
        #This function yeilds "n" number of transaction at time
        for i in range (0, len(transaction),number):
            yield transaction[i:i+2]

    def merkel_tree(self,transactions):
        #Here we will find the merkel tree hash of all
        transactions passed to this fuction
        #Problem is solved using recursion techqiue

        # Given a list of transactions, we concatinate the
        hashes in groups of two and compute
        # the hash of the group, then keep the hash of group.
        We repeat this step till
        # we reach a single hash
        sub_tree=[]
        for i in chunks(transactions,2):
            if len(i)==2:
                hash = sha256(str(i[0]+i[1])).hexdigest()
            else:
```

```
                hash = sha256(str(i[0]+i[0])).hexdigest()
            sub_tree.append(hash)
        # When the sub_tree has only one hash then we reached
        our merkel tree hash.
        #Otherwise, we call this fuction recursively
        if len(sub_tree) == 1:
            return sub_tree[0]
        else:
            return self.merkel_tree(sub_tree)

if __name__=='__main__':
    mk=MerkelTree()
    merkel_hash= mk.merkel_tree(["TX1","TX2","TX3","TX4","TX5",
    "TX6"])
    print merkel_hash
```

Putting It All Together

To get to this section, we covered all the necessary components of blockchain that can help us understand how it really works. After going through them, namely cryptography, game theory, and computer science engineering concepts, we must have developed a notion of how blockchains might work. Though these concepts have been around for ages, no one could ever imagine how the same old stuff can be used to build a transforming technology such as blockchain. Let us have a quick recap of some fundamentals we covered so far, and we will build further understanding on those concepts. So here they are:

- Cryptographic functions are one-way and cannot be inverted. They are deterministic and produce the same output for a given input. Any changes to the input would produce a completely different output when hashed again.

- Using public key cryptography, digital signatures are possible. It helps in verifying the authenticity of the person/entity that has signed. Considering the private key is kept confidential, it is not feasible to forge a signature with someone else's identity. Also, if someone has signed on any document or a transaction, they cannot later deny they did not.
- Using game theoretic principles and best practices, robust systems can be designed that can sustain in most of the odd situations. Systems that can face the Byzantine Generals' Problem need to be handled properly. Our approach to any system design should be such that the participants play by the rules to get the maximum payoff; deviating from the protocol should not really benefit them.
- The blockchain data structure, by using the cryptographic hashes, provides a tamper resistant chain of blocks. The usage of Merkle trees makes the transaction verification easier and faster.

With all these concepts in mind, let us now think of a real blockchain implementation. What problems can you think of that need to be addressed for such a decentralized system to work properly? Well, there are loads of them; some would be generic to most of the blockchain use cases and some would be specific to a few. Let us discuss at least some of the scenarios that need to be addressed:

- Who would maintain the distributed ledger of transactions? Should all the participants maintain, or only a few would do? How about the computing nodes that are not powerful enough to process transactions or do not have enough storage space to accommodate the entire history of transactions?

- How is it possible to maintain a single consistent state of the distributed ledger? Network latency, packet drops, deliberate hacking attempts, etc. are inevitable. How would the system survive all these?
- Who would validate or invalidate the transactions? Would only a few authorized nodes validate, or all the nodes together would reach a consensus? What if some of the nodes are not available at a given time?
- What if some computing nodes deliberately want to subvert the system or try to reject some of the transactions?
- How would you upgrade the system when there is no centralized entity to take the responsibility? In a decentralized network, what if a few computing nodes upgrade themselves and the rest don't?

There are in fact a lot more concerns that need to be addressed apart from the ones just mentioned. For now we will leave you with those thoughts, but most of those queries should be clarified by the end of this chapter.

Let us start with some basic building blocks of a blockchain system that may be required to design any decentralized solution.

Properties of Blockchain Solutions

So far, we have only learned the technical aspects of blockchain solutions to understand how blockchains might work. In this section, we will learn some of the desired properties of blockchains.

Immutability

It is the most desired property to maintain the atomicity of the blockchain transactions. Once a transaction is recorded, it cannot be altered. If the transactions are broadcast to the network, then almost everyone has a copy of it. With time, when more and more blocks are added to the blockchain, the immutability increases and after a certain time, it becomes completely immutable. For someone to alter the data of so many blocks in a series is not practically feasible because they are cryptographically secured. So, any transaction that gets logged remains forever in the system.

Forgery Resistant

A decentralized solution where the transactions are public is prone to different kinds of attacks. Attempts at forgery are the most obvious of all, especially when you are transacting anything of value. Cryptographic hash and digital signatures can be used to ensure the system is forgery resistant. We already learned that it is computationally infeasible to forge someone else's signature. If you make a transaction and sign a hash of it, no one can alter the transaction later and say you signed a different transaction. Also, you cannot later claim you never did the transaction, because it is you who signed it.

Democratic

Any peer-to-peer decentralized system should be democratic by design (may not be fully applicable to the private blockchain, which we will park for later). There should not be any entity in the system that is more powerful than the others. Every participant should have equal rights in any situation, and decisions are made when the majority reaches a consensus.

Double-Spend Resistant

Double-spend attacks are quite common in monetary as well as nonmonetary transactions. In a cryptocurrency setting, a double-spend attempt is when you try to spend the same amount to multiple people. Example: You have $100 in your account and you pay $90 to two or more parties is a type of double-spend. This is a little different when it comes to cryptocurrency such as Bitcoin where there is no notion of a closing balance. Input to a transaction (when you are paying to someone) is the output of another transaction where you have received at least the amount you are paying through this transaction. Assume Bob received $10 from Alice some time back in a transaction. Today if Bob wants to pay Charlie $8, then the transaction in which he received $10 from Alice would be the input to transact with Charlie. So, Bob cannot use the same input (Alice's $10 paid to him) multiple times to pay to other people and double-spend. Just to give you a different example: if someone owns some land and sells the same piece of land to two people.

In a centralized system it is quite easy to prevent double-spend because the central authority is aware of all the transactions. A blockchain solution should also be immune to such double-spend attacks. While cryptography ensures authenticity of a transaction, it cannot help prevent double-spend. Because, technically, both a normal transaction and a double-spend transaction are genuine. So, the only way possible to prevent double-spend is to be aware of all the transactions. If we are aware of all transactions that happened in the past, we can figure out if a transaction is an attempt to double-spend. So, the nodes that would validate the transactions should definitely be accessible to the whole blockchain data since the genesis block.

Consistent State of the Ledger

The properties we just discussed ensure that the ledger is consistent throughout, to some extent. Imagine a situation when some nodes deliberately want a transaction to not go through and to get rejected. Or, if somehow some nodes are not in sync with the ledger and hence not aware of a few transactions that took place while they were offline, then to them a transaction may look like fraudulent. So, how to ensure consensus among the participants is something that needs to be handled very carefully. Recollect the Byzantine Generals' Problem. The right kind of consensus suitable for a given situation plays the most important role to ensure stability of a decentralized solution. We will learn different consensus mechanisms later in this book.

Resilient

The network should be resilient enough to withstand temporary node failures, unavailability of some computing nodes at times, network latency and packet drops, etc.

Auditable

A blockchain is a chain of blocks that are linked together through hashes. Since the transaction blocks are linked back till the genesis block, auditability already exists and we have to ensure that it does not break at any cost. Also, if one wants to verify whether a transaction took place in the past, then such verification should be quicker.

Blockchain Transactions

When we say blockchain, we mean a blockchain of transactions, right? So it starts from a transaction and then the transaction goes through a series of steps and ultimately resides in the blockchain. Since blockchain is a

peer-to-peer phenomenon, if you are dealing with a use case that has a lot of transactions taking place every second, you may not want to flood the whole network with all transactions. Obviously when an individual or an entity is making a transaction, they just have to broadcast it to the whole network. Once that happens, it has to be validated by multiple nodes. Upon validation, it has to again get broadcast to the whole network for the transaction to get included in the blockchain. Now, why not a transaction chain instead of a blockchain? It may make sense to some extent if your business case does not involve a lot of transactions. However, if there are a huge number of transactions every second, then hashing them at transaction level, keeping a trail of it, and broadcasting that to the network can make the system unstable. You may want a certain number of transactions to be grouped in a block and broadcast that block. Broadcasting individual transactions can become a costly affair. Another good reason for a blockchain instead of a transaction chain is to prevent Sybil Attack. In Chapter 3, you will learn in more detail how the PoW mining algorithm is used and one node is chosen at random that could propose a block. If it was not the case, people might create replicas of their own node to subvert the system.

In its most simplified form, the blockchain transactions go through the following steps to get into the blockchain:

- Every new transaction gets broadcast to the network so that all the computing nodes are aware of that fact at the time it took place (to ensure the system is double-spend resistant) .
- Transactions may get validated by the nodes to accept or reject by checking the authenticity.
- The nodes may then group multiple transactions into blocks to share with the other nodes in the network.

- Here comes the difficult situation. Who would propose the block of transactions that they have grouped individually? Broadly speaking, the generation of new blocks should be controlled but not in a centralized fashion, and the mechanism should be such that every node is given equal priority. Every node agreeing upon a block is called the consensus, but there are different algorithms to achieve the same objective, depending on your use case. We will discuss different consensus mechanisms in the following section.
- Though there is no notion of a global time due to network latency, packet drops, and geographic locations, such a system still works because the blocks are added one after another in an order. So, we can consider that the blocks are time stamped in the order they arrive and get added in the blockchain.
- Once the nodes in the network unanimously accept a block, then that block gets into the blockchain and it includes the hash of the block that was created right before it. So this extends the blockchain by one block.

We already discussed the blockchain data structure and the Merkle trees, so we understand their value now. Recollect that when a node would like to validate a transaction, it can do so more efficiently by the Merkle path. The other nodes in the network do not have to share the full block of data to justify proof of membership of a transaction in a block. Technically speaking, memory efficient and computer-friendly data structures such as "Bloom filters" are widely used in such scenarios to test the membership.

Also, note that for a node to be able to validate a transaction, it should ideally have the whole blockchain data (transactions along with their metadata) locally. You should select an efficient storage mechanism that the nodes will adopt based on your use case.

Distributed Consensus Mechanisms

When the nodes are aware of the entire history of transactions by having a local copy of the full blockchain data to prevent double-spend, and they can verify the authenticity of a transaction through digital signatures, what is the use of consensus? Imagine the presence of one or more malicious nodes. Can't they say an invalid transaction is a valid one, or vice versa? Recollect the Byzantine Generals' Problem, which is most likely to occur in many decentralized systems. To overcome such issues, we need a proper consensus mechanism in place.

So far in our discussion, the one thing that is not clear yet is who proposes the block. Obviously, not every node should propose a block to the rest of the nodes at the same time because it is only going to create a mess; forget about the consistent state of the ledger. On the other hand, had it been the case with just transactions without grouping them into blocks, you could argue that if every transaction gets broadcast to the whole network and every node in the network casts a vote on those individual transactions, it would only complicate the system and lead to poor performance.

So, grouping transactions into blocks is important for obvious reasons and consensus is required on a block by block basis. The best strategy for this problem is that only one block should propose a block at a time and the rest of the nodes should validate the transactions in the block and add to their blockchains if transactions are valid. We know that every node maintains its own copy of the ledger and there is no centralized source to sync from. So, if any one node proposes a block and the rest of the nodes agree on it, then all those nodes add that block to their respective blockchains. In such a design, you would prefer that there are at least a few minutes of gap in block creation and it should not be the case where multiple blocks arrive at the same time. Now the question is: who might be that lucky node to propose a block? This is the trickiest part and can lead to proper consensus; we will discuss this aspect under different consensus mechanisms.

These consensus mechanisms actually come from game theory. Your system should be designed such that the nodes get the most benefit if they play by the rules. One of the aspects to ensure the nodes behave honestly is to reward for honest behavior and punish for fraudulent activities. However, there is a catch here. In a public blockchain such as Bitcoin, one can have many different public identities and they are quite anonymous. It gets really difficult to punish such identities because they have a choice to avoid that punishment by creating new identities for themselves. On the other hand, rewarding them works great, because even if someone has multiple identities, they can happily reap the rewards given to them. So, it depends on your business case: if the identities are anonymous, then punishing them may not work, but may work well if the identities are not anonymous. You may want to consider this reward/punish aspect despite having a great mechanism to select a node that would propose the next block. This is because you would never know in advance if the node selected is a malicious node or an honest one. Keep in mind the term *mining* that we may be using quite often, and it would mean generating new blocks.

The goal of consensus is also to ensure that the network is robust enough to sustain various types of attacks. Irrespective of the types of consensus algorithms one may choose depending on the use case, it has to fall into the Byzantine fault tolerant consensus mold to be able to get accepted. Let us now learn some of the consensus mechanisms pertaining to the blockchain scenarios that we may be able to use in different situations.

Proof of Work

The PoW consensus mechanism has been around for a long time now. However, the way it was used in Bitcoin along with other concepts made it even more popular. We will discuss this consensus mechanism at its basic level and look at how it is implemented in Bitcoin in Chapter 3.

The idea behind the PoW algorithm is that certain work is done for a block of transactions before it gets proposed to the whole network. A PoW is actually a piece of data that is difficult to produce in terms of computation and time, but easy to verify. One of the old usages of PoW was to prevent email spams. If a certain amount of work is to be done before one can send an email, then spamming a lot of people would require a lot of computation to be performed. This can help prevent email spams. Similarly, in blockchain as well, if some amount of compute-intensive work is to be performed before producing a block, then it can help in two ways: one is that it will definitely take some time and the second is, if a node is trying to inject a fraudulent transaction in a block, then rejection of that block by the rest of the nodes will be very costly for the one proposing the block. This is because the computation performed to get the PoW will have no value.

Just think about proposing a block without much of effort vs. doing some hard work to be able to propose a block. If it was with almost no effort, then proposing a node with a fraudulent transaction and getting rejected would not have been a big concern. People may just keep proposing such blocks with a hope that one may get through and make it to the blockchain sometime. On the contrary, doing some hard work to propose a block prevents a node from injecting a fraudulent transaction in a subtle way.

Also, the difficulty of the work should be adjustable so that there is a control over how fast the blocks can get generated. You must be thinking, if we are talking about some work that requires some computation and time, what kind of work must it be? It is very simple yet tricky work. An example would help here. Imagine a problem where you have to find a number which, if you hash, the hashed output would start with the alphabet "a." How would you do it? We have learned about the hash functions and know that there are no shortcuts to it. So you would just keep guessing (maybe take any number and keep incrementing by one) the numbers and keep hashing them to see if that fits the bill. If the difficulty level needs to be

increased, then one can say it starts with three consecutive "a"s. Obviously, finding a solution for something like "axxxxxxx" is easier to find compared with "aaaxxxxx" because the latter is more constrained.

In the example just given, if multiple different nodes are working to solve such a computational puzzle, then you will never know which node would solve it first. This can be leveraged to select a random node (this time it is truly random because there is no algorithm behind it) that solves the puzzle and proposes the block. It is extremely important to note that in case of public blockchains, the nodes that are investing their computing resources have to be rewarded for honest behavior, else it would be difficult to sustain such a system.

Proof of Stake

The Proof of Stake (PoS) algorithm is another consensus algorithm that is quite popular for distributed consensus. However, what is tricky about it is that it isn't about mining, but is about validating blocks of transactions. There are no mining rewards due to generation of new coins, there are only transaction fees for the miners (more accurately validators, but we will keep using 'miners' so it gets easier to explain).

In PoS systems, the validators have to bond their stake (mortgage the amount of cryptocurrency thay would like to keep at stake) to be able to participate in validating the transactions. The probability of a validator producing a block is proportional to their stake; the more the amount at stake, the greater is their chance to validate a new block of transactions. A miner only needs to prove they own a certain percentage of all coins available at a certain time in a given currency system. For example, if a miner owns 2% of all Ether (ETH) in the Ethereum network, they would be able to mine 2% of all transactions across Ethereum. Accordingly, who gets to create the new block of transaction is decided, and it varies based on the PoS algorithm you are using. Yes, there are variants of PoS algorithm such as naive PoS, delegated PoS, chain-based PoS, BFT-style PoS, and Casper

PoS, to name a few. Delegated PoS (DPOS) is used by Bitshares and Casper PoS is being developed to be used in Ethereum.

Since the creator of a block in a PoS system is deterministic (based on the amount at stake), it works much faster compared with PoW systems. Also, since there are no block rewards and just transaction fees, all the digital currencies need to be created in the beginning and their total amount is fixed all through.

The PoS systems may provide better protection against malicious attacks because executing an attack would risk the entire amount at stake. Also, since it does not require burning a lot of electricity and consuming CPU cycles, it gets priority over PoW systems where applicable.

PBFT

PBFT is the acronym for the Practical Byzantine Fault Tollerance algorithm, one of the many consensus algorithms that one can consider for their blockchain use case. Out of so many blockchain initiatives, Hyperledger, Stellar, and Ripple are the ones that use PBFT consensus.

PBFT is also an algorithm that is not used to generate mining rewards, similar to PoS algorithms. However, the technicalities in their respective implementations are different. The inner working of PBFT is beyond the scope of this book, but at a high level, requests are broadcast to all participating nodes that have their own replicas or internal states. When nodes receive a request, they perform the computation based on their internal states. The outcome of the computation is then shared with all other nodes in the system. So, every node is aware of what other nodes are computing. Considering their own computation results along with the ones received from ther nodes, they make a decision and commit to a final value, which is again shared across the nodes. At this moment, every node is aware of the final decision of all other nodes. Then they all respond with their final decisions and, based on the majority, the final consensus is achieved. This is demonstrated in Figure 2-24.

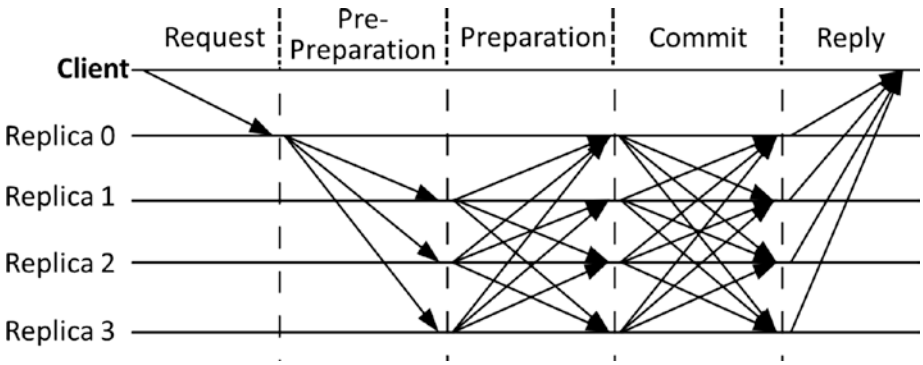


Figure 2-24. *PBFT consensus approach*

PBFT can be efficient compared with other consensus algorithms, based on the effort required. However, anonymity in the system may be compromised because of the way this algorithm is designed. It is one of the most widely used algorithms for consensus even in non-blockchain environments.

Blockchain Applications

While we looked at the nuts and bolts of blockchain throughout this chapter, it is also important that we look at how it is being used in building blockchain solutions. There are applications being built that treat blockchain as a backend database behind a web server, and there are applications that are completely decentralized with no centralized server. Bitcoin blockchain, for example, is a blockchain application where there is no server to send a request to! Every transaction is broadcast to the entire network. However, it is possible that a web application is built and hosted in a centralized web server, and that makes Bitcoin blockchain updates when required. Take a look at Figure 2-25 where a Bitcoin node broadcasts the transactions to the nodes that are reachable at a given point in time.

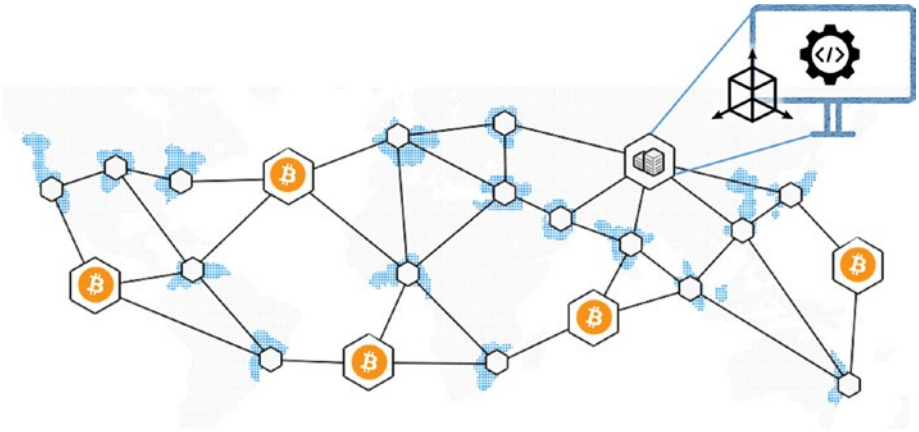


Figure 2-25. *Bitcoin blockchain nodes*

From a software application perspective, every node is self-sufficient and maintains its own copies of the blockchain database. Considering Bitcoin blockchain as a benchmark, the blockchain applications with no centralized servers appear to be the purest decentralized applications and most of them fall under the "public blockchain" category. Usually for such public blockchains, usage of resources from cloud service providers such as Microsoft Azure, IBM Bluemix, etc. are not quite popular yet. For most of the private blockchains, however, the cloud service providers have started to gain popularity. To give you an analogy, there could be one or more web applications for different departments or actors, all of them having their own Blockchin backends and still the blockchains are in sync with each other. In such a setting, though technical decentralization is achieved, politically it could still be centralized. Even though control or governance is enforced, the system is still able to maintain transparency and trust because of the accessibility to single source of truth. Take a look at Figure 2-26, which may resemble most of the blockchain POCs or applications being built on blockchain where blockchains are hosted by some cloud service provider by consuming their blockchain-as-a-Service (BaaS) offering.

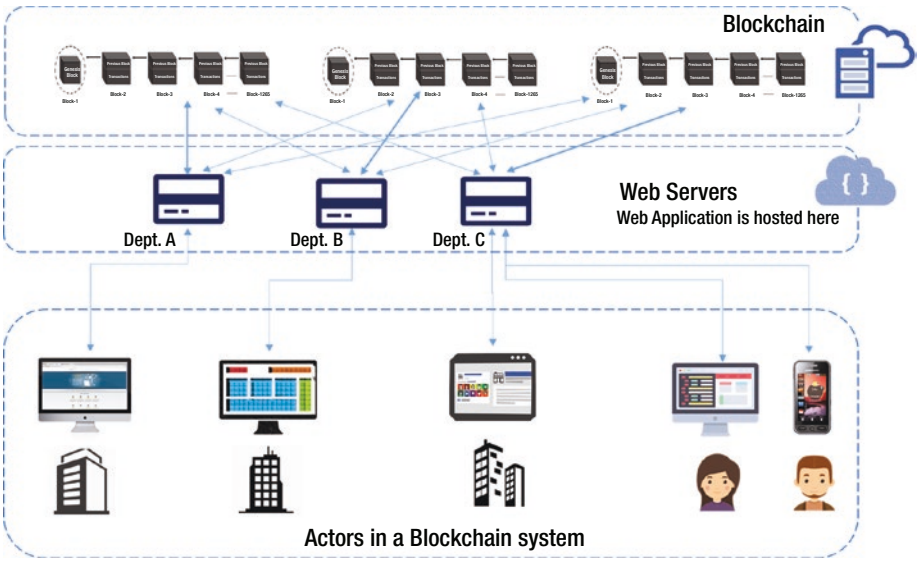


Figure 2-26. *Cloud-powered blockchain system*

It may not be necessary that all the departments have their own different web application. One web application can handle requests from multiple different actors in the system with proper access controm mechanisms. It might be a good idea that all the actors in the system have their own copies of blockchains. Having a local copy of blockchain not only helps maintain transparency in the system, but also may help generate data-driven insights with ready access to data all the time. The different "blockchains" maintained by different actors in the system are consistent by design, thanks to consensus algorithms such as PoW, PoS, etc. Most of the private blockchains prefer any consensus algorithm other than PoW to mitigate heavy resource consumption, and save electricity and computing power as much possible. The PoS consensus mechanism is quite common when it comes to private or consortium blockchains. Since blockchain is disrupting many aspects of businesses, and there was no better way of enabling transparency among them, creating a blockchain solution in the

cloud with a "pay as you use" model is gaining momentum. Cloud services are helping businesses leapfrog in their blockchain-enabled digital transformation journey with minimal upfront investments.

There are also decentralized applications (DApps) being built on Ethereum blockchain networks. These applications could be permissioned on private Ethereum or could be permissionless on a public Ethereum network. Also, these applications could be for different use cases on the same public Ethereum network. Though we will cover the Ethereum-specific details later in this book, just look at Figure 2-27 for a high-level understanding of how those applications might look.

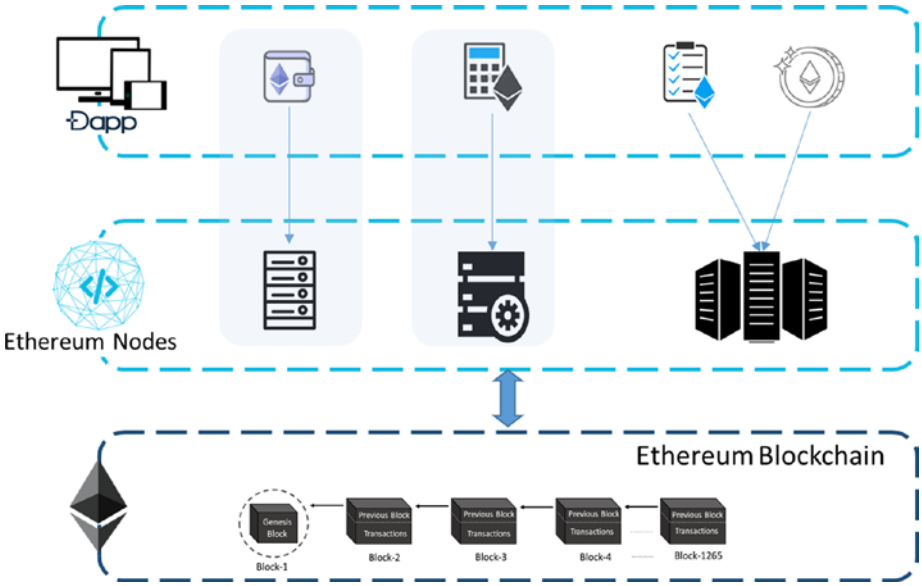


Figure 2-27. *DApps on Ethereum network*

As discussed already in previous sections, developing blockchain applications is only limited by your imagination. Pure blockchain native applications could be built. Applications that treat blockchain as just a backend are also being built, and there are hybrid applications that are also being built that use the legacy applications and use blockchain for

some specific purpose only. So far, blockchain scalability is one of the biggest concerns. Though the scalability itself is in research, let us learn some of the scalability techniques.

Scaling Blockchain

We looked at blockchain from a historic perspective and how it proves to be one of the most disruptive technologies as of today. While exploring it technically in this chapter, we learned about the scalability issues inherent to most of the Blockchin flavors. By design, blockchains are difficult to scale and thus a research area in academia and for some innovation-driven corporates. If you look at the Bitcoin adoption, it is not being used to replace fiat currencies due to the inherent scalability challenges. You cannot buy a coffee using Bitcoin and wait for an hour for the transaction to settle. So, Bitcoins are being used as an asset class for investers to invest in. A Bitcoin blockchain network is not capable of accommodating as many transactions as that of Visa or MasterCard, as of today.

Recollect the consensus protocols we have studied so far, such as PoW of Bitcoins or Ethereum, or PoS and other BFT consensus of some other blockchain flavors such as Multichain, Hyperledger, Ripple, or Tendermint. All of these consensus algorithms' primary objective is Byzantine fault tolerance. By design, every node (at least the full nodes) in a blockchain network maintains its own copy of the entire blockchain, validates all transactions and blocks, serves requests from other nodes in the network, etc. to achieve decentralization, which becomes a bottleneck for scalability. Look at the irony here–we add more servers in a centralized system for scalability, but the same does not apply in a decentralized system because with more number of nodes, the latency only increases. While the level of decentralization could increase with a greater number of nodes in a decentralized network, the number of transactions in the network also increases, which leads to increased requirements of

computing and storage resources. Keep in mind that this situation is applicable more on public blockchains and less so for private blockchains. Private blockchains could easily scale compared with the public ones because the controlling entities could define and set node specifications with high computation power and more bandwidth. Also, there could be certain tasks offloaded from blockchain and computed off-chain that could help the system scale well.

In this chapter, we will learn some of the generic scaling techniques, and discuss Bitcoin- and Ethereum-specific scaling techniques in their respective chapters. Please keep in mind that all scaling techniques may not apply to all kinds of blockchain flavors or use cases. The best way is to understand the techniques technically and use the best possible one in a given situation.

Off-Chain Computation

Off-chain computation is one of the most promising techniques to scale blockchain solutions. The idea is to limit the usage of blockchain and do the heavy lifting outside of it, and only store the outcomes on blockchain. Keep in mind that there is no standard definition of how the off-chain computation should happen. It is heavily dependent on the situation and the people trying to address it. Also, different blockchain flavors may require different approaches for off-chain computation. At a high level, it is like another layer on top of blockchain that does heavy, compute-intensive work and wisely uses the blockchain. Obviously, you may not be able to retain all the characteristics of blockchain by doing computations off-chain, but it is also true that you may not need blockchain for all kinds of computing requirements and may use it only for specific pain points.

The off-chain computations could be on a sidechain, could be distributed among a random group of nodes, or could be centralized as well. The side chains are independent of the main blockchain. It not only helps scale the blockchain well, it also isolates damages to the sidechain

and prevents the main blockchain from any damages from a sidechain. One such example sidechain is the "Lightning Network" for Bitcoins that should help in faster execution of transactions with minimal fee; that will support micropayments as well. Another example of a sidechain for Bitcoins is "Zerocash," whose primary objective is not really scalability, but privacy. If you are using Zerocash for Bitcoin transactions, you cannot be tracked and your privacy is preserved. We will limit our discussion to the generic scalability techniques and not get into a detailed discussion of Bitcoin scalability in this book.

One obvious question that might come up at the moment is how people would check the authenticity of the transactions if they are sent off-chain. First, to create a valid transaction, you do not need a blockchain. We learned in the "Cryptography" section in this chapter about the assymetric key cryptography that is used by the blockchain system. To make a transaction, you have to be the owner of a private key so you can sign the transaction. Once the transaction is created, there are advantages when it gets into the blockchain. Double-spend is not possible with Bitcoin blockchain, and there are other advantages, too. For now, the only objective is to get you on board with the fact that you can create a transaction as long as you own the private key for your account.

Bitcoin blockchains are a stateless blockchain, in the sense that they do not maintain the state of an account. Everything in Bitcoin blockchain is present in the form of a transaction. To be able to make a transaction, you have to consume a previous transaction and there is no notion of "closing balance" for an account, as such. On the contrary, Ethereum blockchain is a "stateful" one! The blocks in Ethereum blockchain contain information regaring the state of the entire block where account balance is also a part. The state information takes up significant space when every node in the network maintains it. This situation is valid for other blockchains as well that are stateful.

Let's take an example to understand this better. Alice and Bob are two parties having multiple transactions between each other. Let's say they usually have 50 monetary transactions in a month. In a stateful blockchain, all these individual transactions would have their state information, and that will be maintained by all the nodes. To address this challenge, the concept of "state channels" is introduced. The idea is to update the blockchain with the final outcome, say, at the end of the month or when a certain transaction threshold is reached, and not with each and every transaction.

State channels are essentially a two-way communication channel between users, objects, or services. This is done with absolute security by using cryptographic techniques. Just to get a heads-up on how it works, take a look at Figure 2-28.

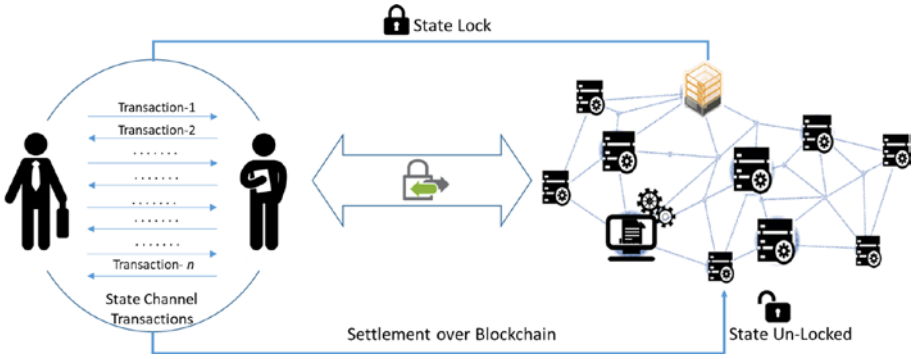


Figure 2-28. *State channels for off-chain computation*

Notice that the off-chain state channels are mostly private and confined among a group of participants. Keep in mind that the state of blockchain for the participants needs to be locked as the first step. Either it could be a MultiSig scheme or a smart contract-based locking. After locking, the participants make transactions among each other that are cryptographically secured. All transactions are cryptographically signed, which makes them verifiable and these transactions are not immediately

submitted to the blockchain. As discussed, these state channels could have a predefined lifespan, or could be bound to the amount of transactions being carried out in terms of volume/quantity or any other quantifiable measure. So, the final outcome of the transactions gets settled on the blockchain and that unlocks the state as the final step.

State channels could be very differently implemented in different use cases, and their implementations are actually left to the developers. It is certainly a way forward and is one of the most critical components for mainstream adoption of blockchain applications. For Bitcoin, the Lightning Network was designed for off-chain computation and to make the payments transaction faster. Similarly, the "Raiden Network" was designed for Ethereum blockchain. There are many other such developments to make micropayments faster and more feasible on blockchain networks.

Sharding Blockchain State

Sharding is one of the scalability techniques that has been there for ages and has been a more sought-after topic for databases. People used this technique differently in different use cases to address specific scalability challenges. Before we understand how it could be used in scaling blockchain as well, let us first understand what it means.

Disk read/write has always been a bottleneck when dealing with huge data sets. When the data is partitioned across multiple disks, the read/write could be performed in parallel and latency decreases significantly. This technique is called sharding. Take a look at Figure 2-29.

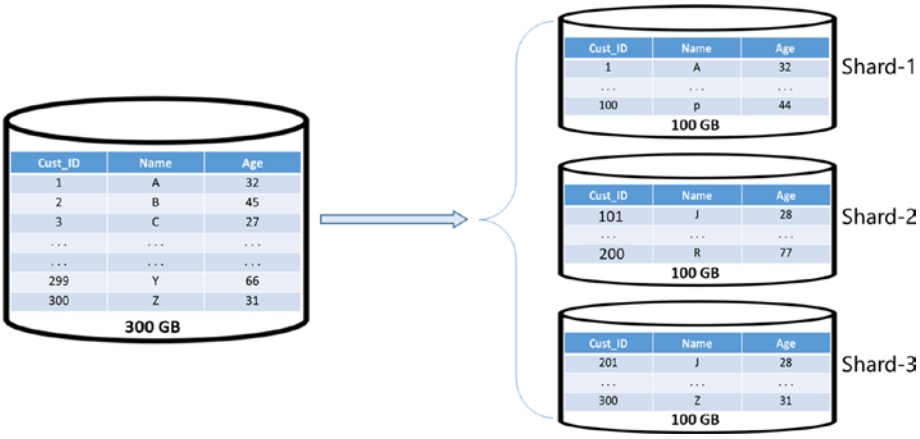


Figure 2-29. *Database sharding example*

Notice in Figure 2-29 how horizontal partitioning is done to distribute a 300GB database table into three shards of 100GB each and stored on separate server instances. The same concept is also applicable for blockchain, where the overall blockchain state is divided into different shards that contain their own substates. Well, it is definitely not as easy as sharding a database with just doing horizontal partitioning.

So, how does sharding really work in the context of blockchain? The idea is that the nodes wouln't be required to download and keep a copy of the entire blockchain. Instead, they would download and keep the portions (shards) relevant to them. By doing so, they get to process only those transactions that are relevant to the data they store, and parallel execution of transactions is possible. So, when a transaction occurs, it is routed to only specific nodes depending on which shards they affect. If you look at it from a different lens, all the nodes are not required to do all sorts of calculations and verifications for each and every transaction. A mechanism or a protocol could be defined for communication between shards when more than one shard is required to process any specific

transactions. Please keep in mind that different blockchains might have different variants of sharding.

To give you an example, you might choose a specific sharding technique for a given situation. One example could be where shards are required to have multiple unique accounts in them. In other words, each unique account is in one shard (more applicable for Ethereum style blockchains that are stateful), and it is very easy for the accounts in one shard to transact among themselves. Obviously, one more level extraction at a shard level is required for sharding to work, and the nodes could keep only a subset of the information.

Summary

In this chapter, we took a deep dive into the core fundamentals of cryptography, game theory, and computer science engineering. The concepts learned would help you design your own blockchain solution that may have some specific needs. Blockchain is definitely not a silver bullet for all sorts of problems. However, for the ones where blockchain is required, it is highly likely that different flavors of blockchain solutions would be needed with different design constructs.

We learned different cryptographic techniques to secure transactions and the usefulness of hash functions. We looked at how game theory could be used to design robust solutions. We also learned some of the core computer science fundamentals such as blockchain data structure and Merkle trees. Some of the concepts were supplimented with example code snippets to give you a jump start on your blockchain assignments.

In the next chapter, we will learn about Bitcoin as a blockchain use case, and how exactly it works.

References

New Directions in Cryptography

Diffie, Whitfield; Hellman, Martin E., "New Directions in Cryptography," IEEE Transactions on Information Theory, Vol IT-22, No 6, https://ee.stanford.edu/~hellman/publications/24.pdf, November, 1976.

Kerckhoff's Principle

Crypto-IT Blog, "Kerckhoff's Principle," www.crypto-it.net/eng/theory/kerckhoffs.html.

Block Cipher, Stream Cipher and Feistel Cipher

http://kodu.ut.ee/~peeter_l/teaching/kryptoi05s/streamkil.pdf.

www.cs.utexas.edu/~byoung/cs361/lecture45.pdf.

www.cs.man.ac.uk/~banach/COMP61411.Info/CourseSlides/Wk2.1.DES.pdf.

https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture3.pdf.

Digital Encryption Standard (DES)

www.facweb.iitkgp.ernet.in/~sourav/DES.pdf.

Advanced Encryption Standard (AES)

www.facweb.iitkgp.ernet.in/~sourav/AES.pdf.

AES Standard Reference

National Institute of Standards and Technology (NIST), "Announcing the Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publication 197*, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, November 26, 2001.

Secured Hash Standard

National Institute of Standards and Technology (NIST), "Announcing the Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publication 197*, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, November 26, 2001.

SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions

NIST, "Announcing DraftFederl Information Processing Standard (FIPS) 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, and Draft Revision of the Applicability Clause of FIPS 180-4, Secure Hash Standard, and Request for Comments," https://csrc.nist.gov/News/2014/Draft-FIPS-202,-SHA-3-Standard-and-Request-for-Com, May 28, 2014.

SHA-3

Paar, Christof, Pelzl, Jan, "SHA-3 and the Hash Function Keccak," *Understanding Cryptography—A Textbook for Students and Practitioners*, (Springer, 2010), https://pdfs.semanticscholar.org/8450/06456ff132a406444fa85aa7b5636266a8d0.pdf.

RSA Algorithm

Kaliski, Burt, "The Mathematics of the RSA Public-Key Cyptosystem," RSA Laboratories, www.mathaware.org/mam/06/Kaliski.pdf.

Milanov, Evgeny, "The RSA Algorithm," https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf. June 3, 2009.

Game Theory

Pinkasovitch, Arthur, "Why Is Game Theory Useful in Business?," *Investopedia*, www.investopedia.com/ask/answers/09/game-theory-business.asp, December 19, 2017.

Proof of Stake Algorithm

Buterin, Vitalik, "A Proof of Stake Design Philosophy," *Medium*, https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51, December 30, 2016.

Ray, James, "Proof of Stake FAQ," *Ethereum Wiki*, https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

Enabling blockchain Innovations with Pegged Sidechains

Back, Adam, Corallo, Matt, Dash Jr, Luke, et al., "Enabling blockchain Innovations with Pegged Sidechains," https://blockstream.com/sidechains.pdf.