**Understanding the**

**Science Behind**

**Blockchain: Cryptography**

The reason you are reading this book is because you want to understand what a

blockchain is, how it works, and how you can write smart contracts on it to do cool

things. And while I perfectly understand that you are excited to get started in this first

chapter, we need to take a step back and look at one fundamental technology that makes

blockchain possible: *cryptography*.

In this chapter, I will explain what cryptography is, the different types of

cryptographic algorithms, how they work, and how they play a vital role in the world

of blockchain. I will also show you how to experiment with the various cryptographic

algorithms using the Python programming language. Even if you are familiar with

cryptography, I suggest scanning through this chapter so that you have a firm foundation

for the subsequent chapters.

**What Is Cryptography?**

Whether you are trying to build a web application to store users’ credentials or writing

a network application to securely transmit encrypted messages, or even trying to

understand how blockchain works, you need to understand one important topic:

*cryptography*.

So, what exactly is cryptography? Put simply, cryptography (or cryptology) is the

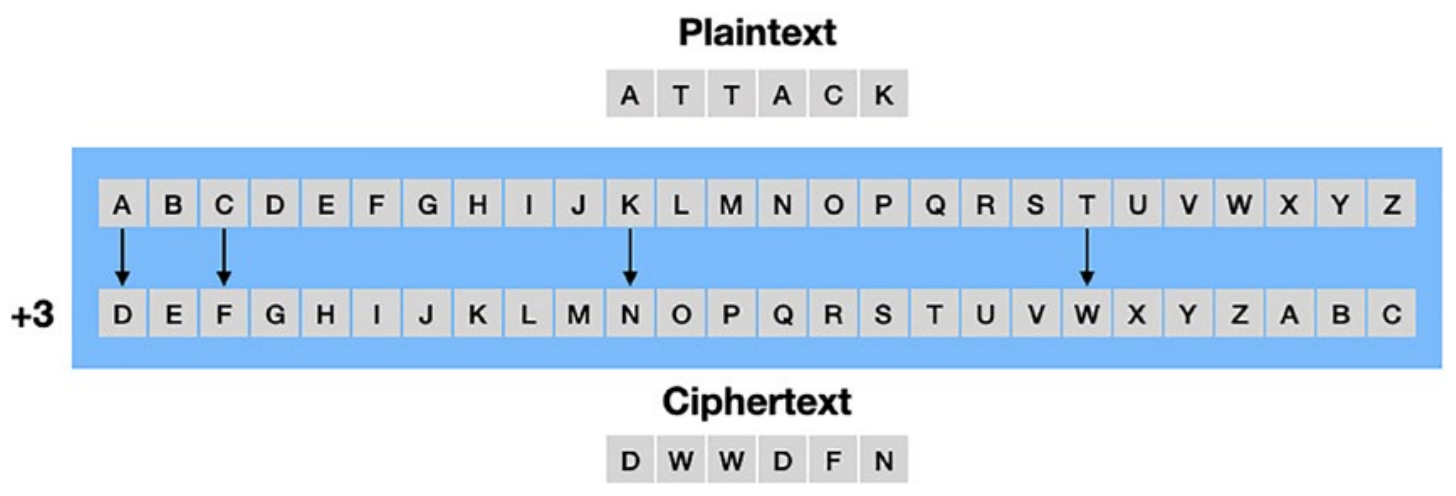
practice and study of hiding information. It is the science of keeping information secret

and safe.

One of the simplest and most widely known cryptographic algorithms is the **Caesar**

**Cipher**. It is a very simple algorithm in which each letter in the plaintext is replaced by a letter

a fixed number of positions down the alphabet. Consider the example shown in Figure 1-1.



As you can observe, each character in the alphabet is shifted down three positions.

A becomes D, B becomes E, and so on. If you want to send a sentence (known as the

plaintext), say ATTACK, to your recipient, you map each of the characters in the sentence

using the above algorithm and derive the encrypted sentence (known as the ciphertext):

DWWDFN. When the recipient receives the ciphertext, they reverse the process to obtain

the plaintext. While this algorithm may seem impressive (especially in the early days

of cryptography), it no longer works as intended as soon as someone knows how the

messages are encrypted. Nevertheless, this is a good illustration of the attempt by early

inventors of cryptography to hide information. Today, the cryptographic algorithms we

use are much more sophisticated and secure.

In the following sections, I will explain the main types of cryptographic functions

and how they are used.

**Types of Cryptography**

There are three main types of cryptography:

• Hash functions

• Symmetric cryptography

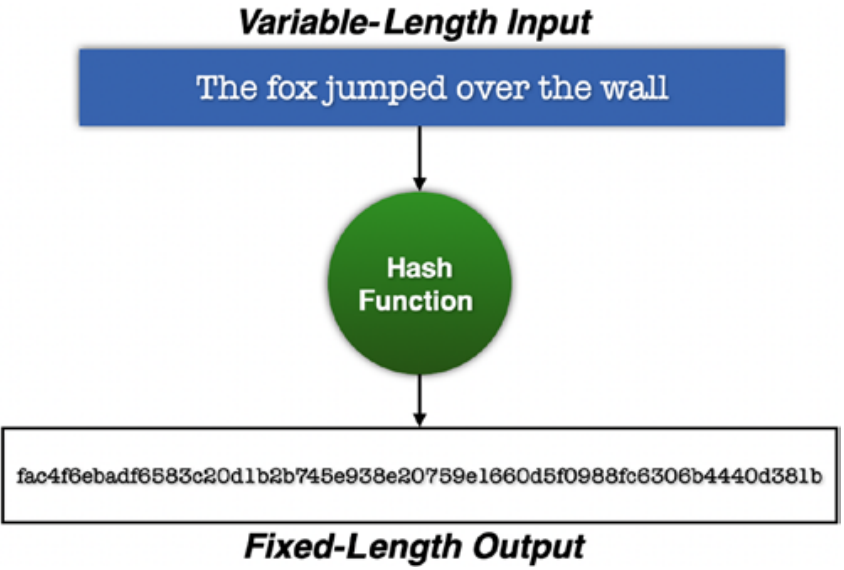
• Asymmetric cryptography

In the following sections, I will go through each of the above types in more detail.

**Hash Functions**

*Hashing* is the process in which you convert a block of data of arbitrary size to a

fixed-size value. The function that performs this process is known as a *hash function* .



***Figure 1-2.*** *A hash function converts a block of data of variable length to*

*a fixed-length output*

**Tip** A commonly-used hash function is **SHA256.** SHA stands for Secure Hash

Algorithms.

For example, the **SHA256** hash function converts a block of text into a 256-bit

hash output. The resultant hash is usually written in hexadecimal, and since each

hexadecimal takes up 4 bits, a 256-bit hash will have 64 characters. To experience how

hashing works, go to <https://emn178.github.io/online-tools/sha256.html>, type in a

sentence, and observe the result

Hashing has the following important properties:

• **Preimage resistant**: Based on the hash created, you cannot obtain

the original block of text.

• **Deterministic**: The same block of text will always produce the same

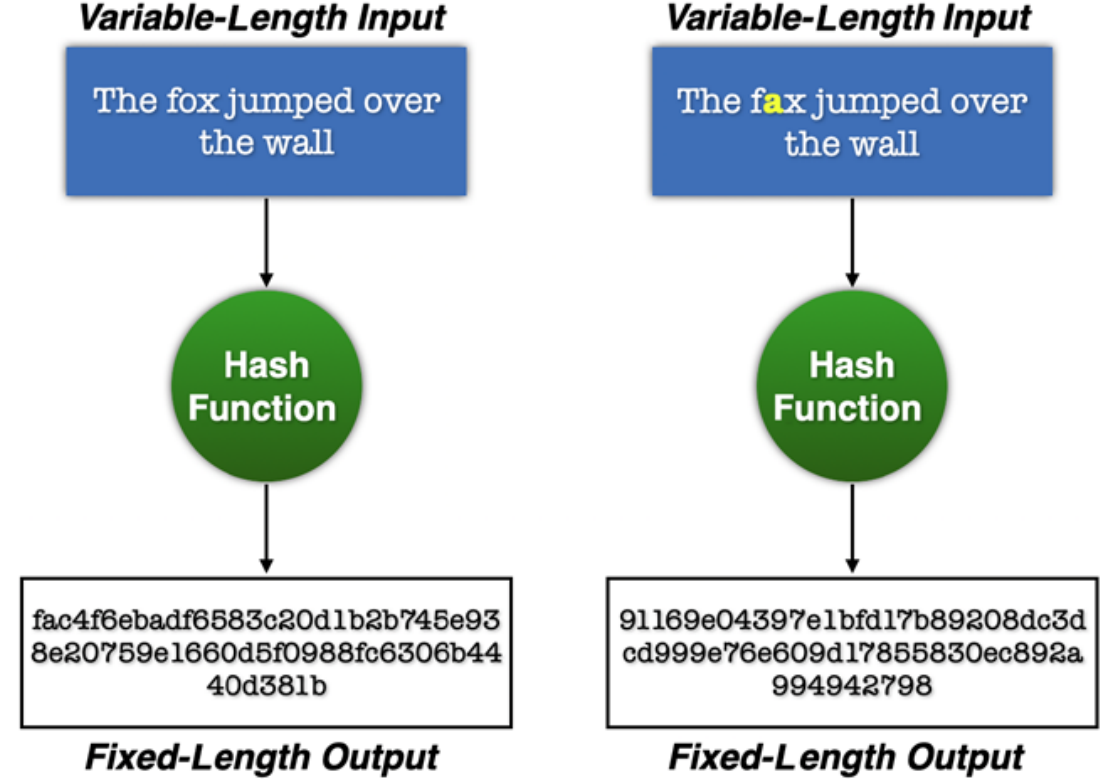
hash output.

• **Collision resistant**: It is hard to find two different blocks of text that

will produce the same hash.

• Another important feature of hashing is that a single change in the original text will

cause a totally different hash to be generated. This is also known as the ***avalanche effect***.



***Figure 1-4.*** *A single change in the input will cause a totally different output hash*

**Uses of Hashing**

Hashing fulfils some very important roles in computing. For one, websites use hashing

to store your password, instead of storing it in plaintext. Storing your password as hashes

prevents hackers from reversing the hashes and obtaining your original password (which

may very likely be used on other websites as well).

Hashing also plays a very crucial role in blockchain, where each block is “chained”

to the previous block using the hash of the previous block. Any modifications to a block

will invalidate the hash stored in the next block, and the rest of the blocks will hence be

invalid.

**Tip** Some commonly used hashing algorithms are MD5, SHA256, SHA512, and

Keccak-256.

**Implementing Hashing in Python**

In Python, you can use the hashlib module to perform hashing. The following code

snippet uses the sha256() function to perform hashing on a string:

import hashlib  
  
# Note that the string to be hashed must be passed to the sha256() function as a byte array. And so you use the bytes() function to convert the string into a byte array.

result = hashlib.sha256(bytes("The quick brown fox jumps over the lazy dog",'utf-8'))  
  
# The sha256() function returns a sha256 hash object. To get the resultant hash in hexadecimal, you can call the hexdigest() function of the sha256 hash object:  
print(result.hexdigest())  
# Alternatively, in Python, you can prefix the string with a b to denote a bytes string literal

result = hashlib.sha256(b"The quick brown fox jumps over the lazy dog")  
print(result.hexdigest())  
  
# If you make a small change to the original string, the output is drastically different from the previous hash:

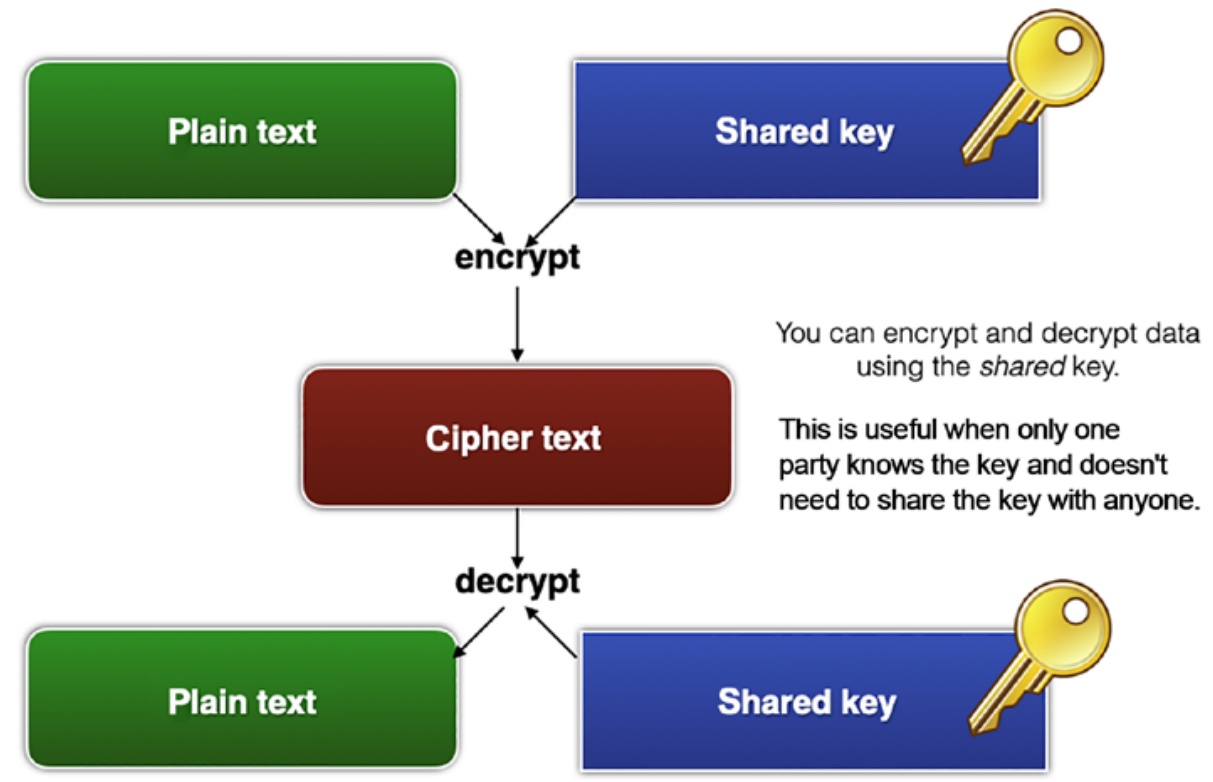
result = hashlib.sha256(b'The quick brown fox jumps over the lazy dag')  
print(result.hexdigest())

**Symmetric Cryptography**

In symmetric cryptography, you use the same cryptographic key (commonly referred to

as the *shared key*) for both the encryption of plaintext and the decryption of ciphertext.

Figure 1-5 shows the use of the shared key for both encryption and decryption.



***Figure 1-5.*** *Using a shared key for encryption and decryption*

Symmetric cryptography is fast and simple, but the main problem is how to ensure

that the key is kept secret. For example, if Tom wants to send a secret message to Susan,

Tom can encrypt the message using the shared key and Susan can decrypt the encrypted

message using the same shared key. The problem here is how can Tom securely send

Susan the shared key? Can Tom email Susan? Send it through SMS or WhatsApp? How

about through the traditional post office? All these methods are not absolutely safe and

are subject to eavesdropping. Moreover, there is this popular saying, “Three may keep a

secret if two of them are dead.” This means, if more than one person knows the secret, it

is no longer a secret.

Having said that, symmetric cryptography has its uses and applications. It is useful

when you want to protect your private data. For example, say you have some confidential

data on your computer that you want to prevent others from seeing. Using symmetric

cryptography, you can encrypt and decrypt the data using the same key, which is only

known to you and no one else.

**Generating the Shared Key in Python**

# pip install cryptography  
  
from cryptography.fernet import Fernet  
  
# Tip The Fernet class is an implementation of symmetric (also known as “secret key”) authenticated cryptography.  
# Fernet uses the AES algorithm in CBC mode with a 128-bit key for encryption.  
  
# generate the shared key  
shared\_key = Fernet.generate\_key()  
print(shared\_key)  
  
# base64 encoded, binary format  
# A new key is generated each time you run this block of code  
# e.g. b'ixXEfrz2NTJlxy1OhxXlsCiFf0Ycg\_GL0Cy0MlgTv4U='  
# For more details, refer to https://github.com/fernet/spec/blob/master/Spec.md  
  
# The generate\_key() function returns a shared key in binary format and it is base64 encoded.

**Performing Symmetric Encryption**

To encrypt your data using the shared key, you first create an instance of the Fernet class using the shared key:

# To encrypt your data using the shared key, you first create an instance of the Fernet class using the shared key:  
  
# create an instance of the Fernet class  
fernet = Fernet(shared\_key)  
# You can then use the encrypt() function to encrypt your data:  
# encrypt the message with the shared key  
ciphertext = fernet.encrypt(  
 bytes("Secret message!", 'utf-8'))  
# remember to pass in a byte array  
  
# You can save the encrypted data into a file:  
# write the encrypted message to file  
with open('message.encrypted', 'wb') as f:  
 f.write(ciphertext)  
  
# And you can save the shared key to file:  
# write the shared key to file  
with open('symmetric\_key.crypt', 'wb') as f:  
 f.write(shared\_key)

**Performing Symmetric Decryption**

# Decryption is similar to encryption. First, load the shared key from the file (which you saved previously):

with open('symmetric\_key.crypt', 'rb') as f:  
 shared\_key = f.read()  
print(shared\_key)  
# Then, create an instance of the Fernet class using the shared key and call the decrypt() function to decode the ciphertext:  
# create an instance of the Fernet class  
fernet = Fernet(shared\_key)  
# decrypt the encrypted message read from file  
with open('message.encrypted', 'rb') as f:  
 print(fernet.decrypt(f.read()).decode("utf-8"))

# 参考资料

*Beginning Ethereum Smart Contracts Programming, Second Edition*!