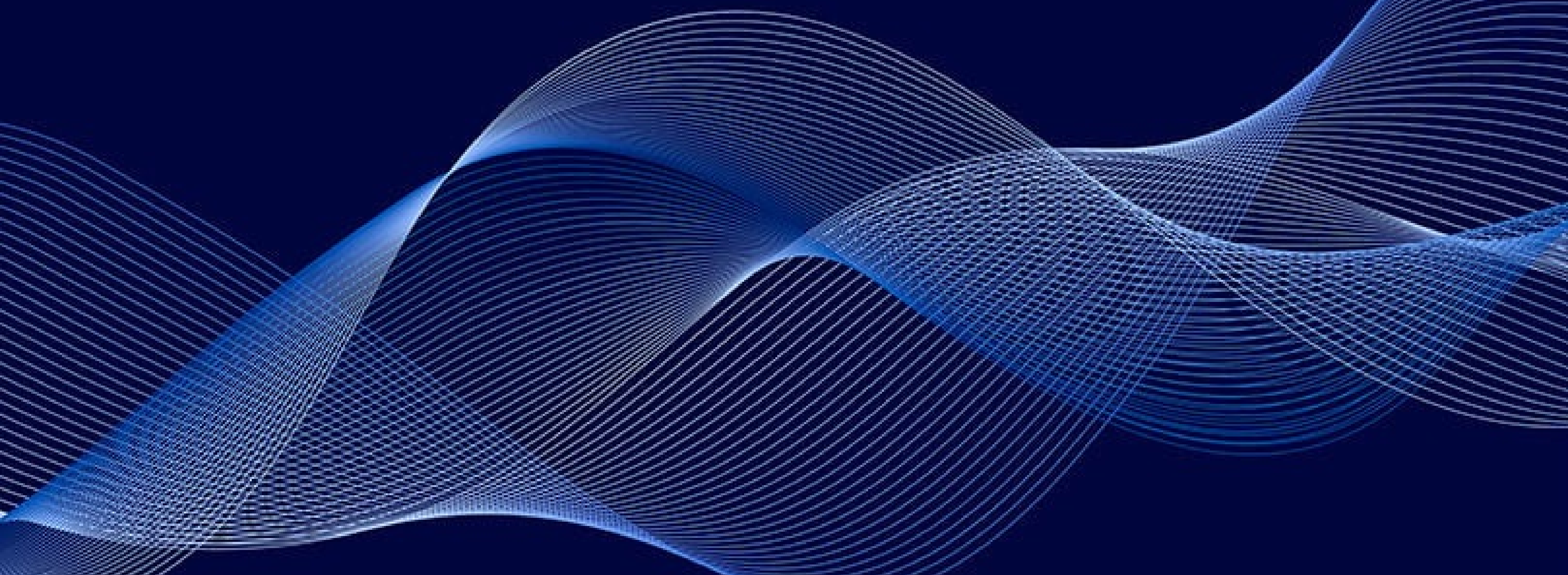
EXPERT INSIGHT

Early Access

Cryptography Algorithms

Build new algorithms in encryption, blockchain, quantum, zero-knowledge, and homomorphic algorithms

Second Edition



Massimo Bertaccini

‹packt›

Cryptography Algorithms

Early Access Publication: Cryptography Algorithms

Early Access Production Reference: B21272

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83508-003-0

Table of Contents

Cryptography Algorithms, Second Edition: Build new algorithms in encryption, blockchain, quantum, zero-knowledge, and homomorphic algorithms

1 Deep Dive into Cryptography

This chapter provides an introduction to cryptography, what it is needed for, and why it is so important in IT. This chapter also gives a panoramic view of the principal algorithms from the history of cryptography, from the Caesar algorithm to the Vernam cipher and other lesser-known algorithms, such as the Beale cipher. Then, **Rivest-Shamir-Adleman** (**RSA**), Diffie–Hellman, and other famous algorithms will be described in detail in the proceeding part of this book. Finally, this chapter will give you the instruments to learn cryptography and the pillars of security conservation.In this chapter, we will cover the following topics:

- A brief introduction to cryptography
- Basic definitions and principal mathematical notations used in the book
- Binary conversion and ASCII code
- Fermat Last's Theorem, prime numbers, and modular mathematics
- The history of the principal cryptographic algorithms and an explanation of some of them (Rosetta cipher, Caesar, ROT13, Beale, Vernam)
- Security notation (semantic, provable, OTP, and so on)

An introduction to cryptography

One of the most important things in cryptography is to understand definitions and notations. I have never been a fan of definitions and notations, first of all, because I am the only one to use notations that I've invented. But I realize that it is very important, especially when we are talking about something related to mathematics, to agree among ourselves. Thus, in this section, I will introduce basic information and citations relating to cryptography.We start with a definition of an algorithm.In mathematics and computer science, an **algorithm** is a finite sequence of well-defined computer-implementable instructions.An important question is: what is a cipher?A **cipher** is a system of any type able to transform plaintext (a message) into not-intelligible text (a ciphertext or cryptogram):

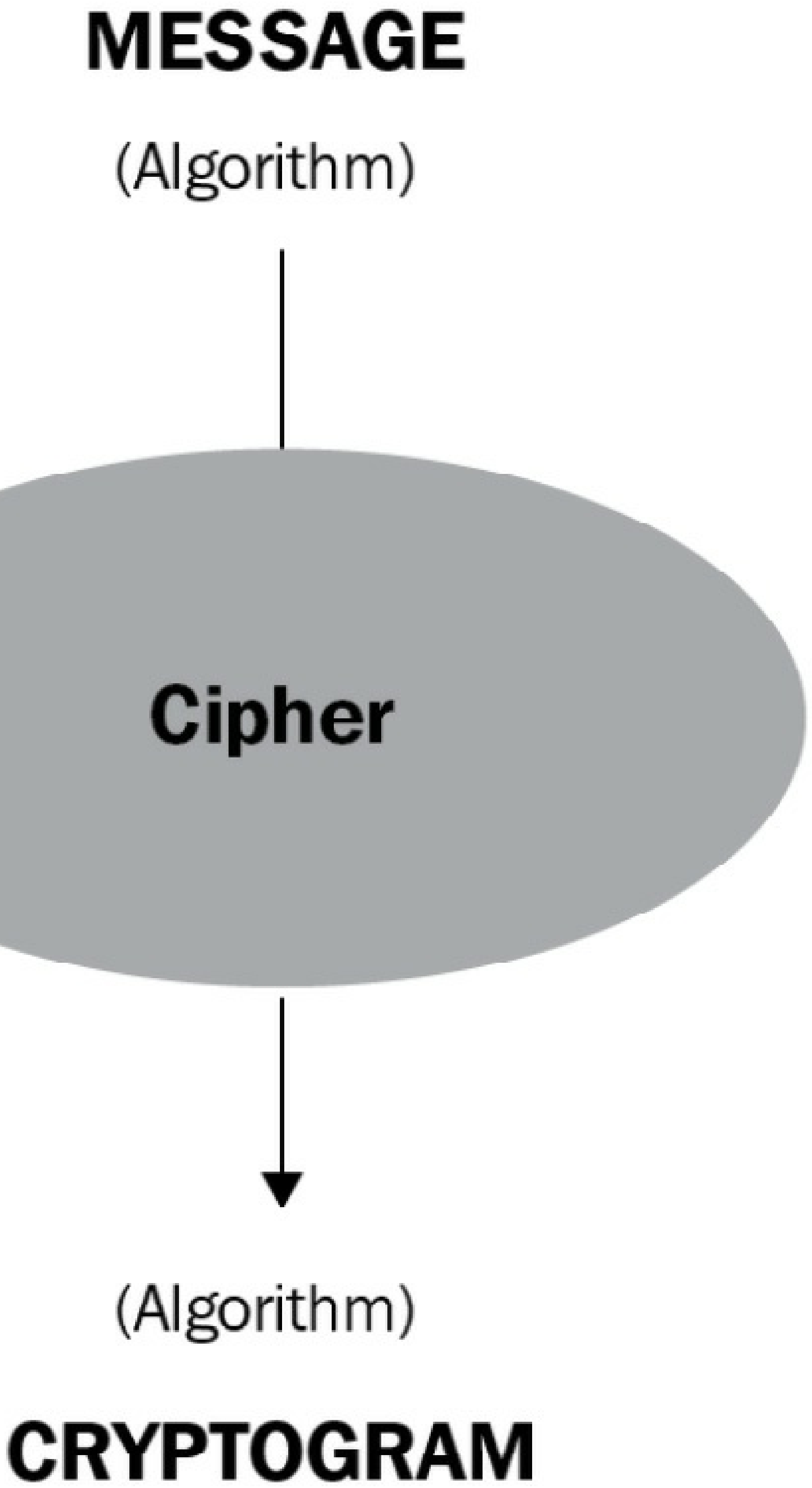

MESSAGE
(Algorithm)
Cipher
(Algorithm)
CRYPTOGRAM

Figure 1.1 – Encryption process

To get some utility from a cipher, we have to set up two operations: encryption and decryption. In simpler terms, we have to keep the message secret and safe for a certain period of time.We define **M** as the set of all the messages and **C** as the set of all the cryptograms.**Encryption** is an operation that transforms a generic message, **m**, into a cryptogram, **c**, applying a function, **E**: `m ------- > f(E) --------- > c` **Decryption** is an operation that returns the message in cleartext, **m**, from the cryptogram, **c**, applying a function, **D**: `c ------- > f(D) --------- > m` Mathematically, **D(E(M))= M**.This means that the **E** and **D** functions are the inverse of each other, and the **E** function has to be injective. **Injective** means different **M** values have to correspond to different **C** values.Note that it doesn't matter whether I use capital letters or lowercase, such as **(M)** or **(m)**; it's inconsequential at the moment. For the moment, I have used round brackets indiscriminately, but later I will use square brackets to distinguish secret elements of a function from known ones, for which I will use square brackets. So, the secret message **M** will be written as **[M]**, just like any other secret parameter. Here, just showing how the algorithms work is within our scope; we'll leave their implementation to engineers.There is another important notation that is key to encryption/decryption. To encrypt and decrypt a message, it is necessary to set up a key. In cryptography, a key is a parameter that determines the functional output of a cryptographic algorithm or cipher. Without a key, the algorithm would produce no useful results.We define **K** as the set of all the keys used to encrypt and decrypt **M**, and **k** as the single encryption or decryption key, also called the session key. However, these two ways to define a key (a set of keys is **K** and a single key is **k**) will always be used, specifying what kind of key it is (private or public).Now that we understand the main concepts of cryptographic notation, it is time to explain the difference between private and public keys:

- In cryptography, a private or secret key (**Kpr**), denoted as **[K]** or **[k]**, is an encryption/decryption parameter known only to one, both, or multiple parties in order to exchange secret messages.
- In cryptography, a public key (**Kpu**) or **(K)** is an encryption key known by everyone who wants to send a secret message or authenticate a user.

So, what is the main difference between private and public keys?The difference is that a private key is used both to encrypt and/or decrypt a message, while a public key is used only to encrypt a message and verify the identity (digital signatures) of humans and computers. This is a substantial and very important issue because it determines the difference between symmetric and asymmetric encryption. Let's give a generic definition of these two methods of encryption:

- **Symmetric encryption** uses only one shared key to both encrypt and decrypt the message.
- **Asymmetric encryption** implements more parameters to generate a public key (to encrypt the message) and just one private key to decrypt the message.

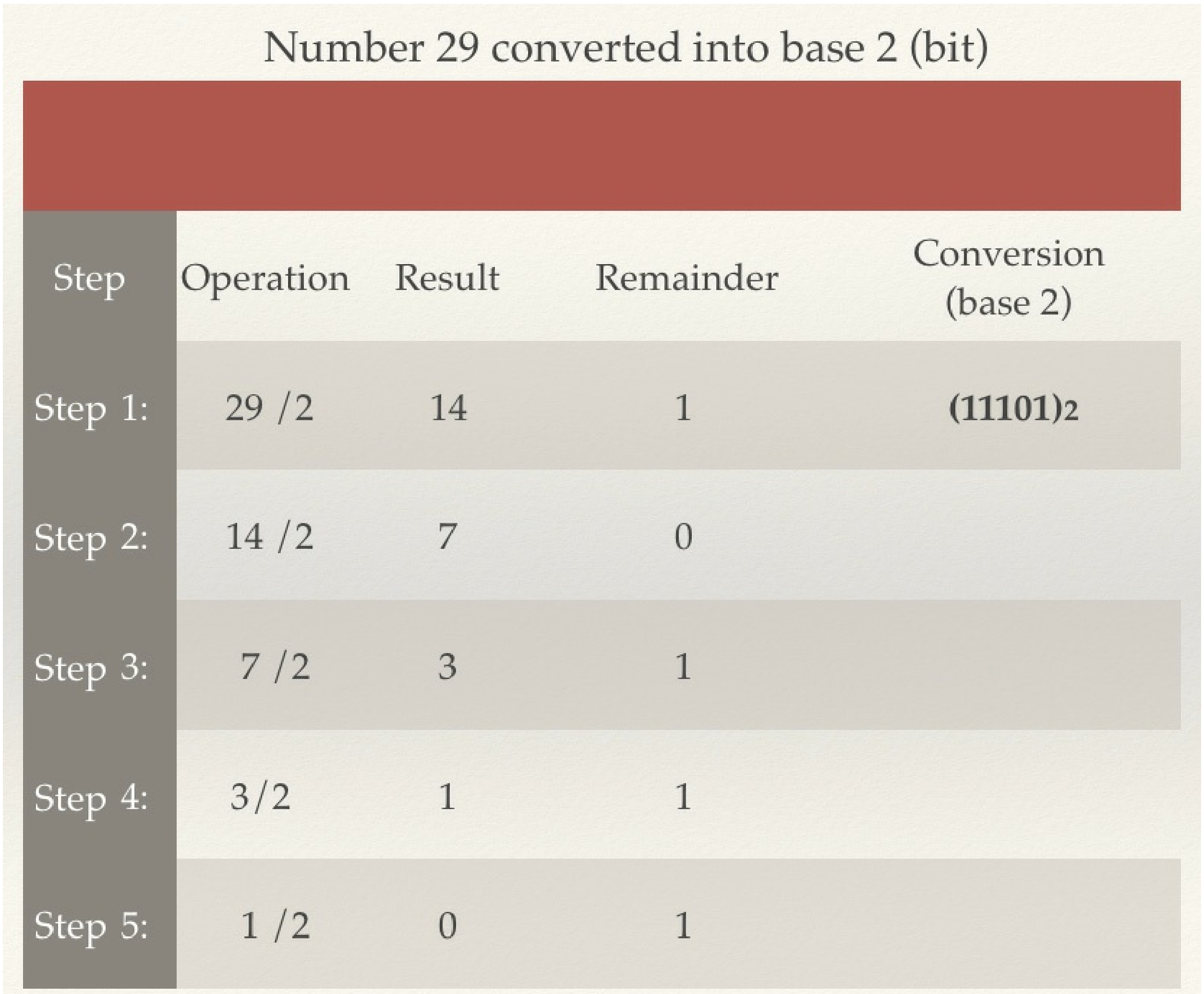
As we will see later on, private keys are used in symmetric encryption to encrypt/decrypt the message with the same key and in asymmetric encryption in a general way for decryption, whereas public keys are used only in asymmetric encryption to encrypt the message and to perform digital signatures. You will see the function of these two types of keys later, but for now, keep in mind that a private key is used both in symmetric and asymmetric encryption, while a public key is used only for asymmetric encryption. Note that it's not my intention to discuss academic definitions and notation, so please try to figure out the scope and the use of each element.One of the main problems in cryptography is the transmission of the key, or the key exchange. This problem resulted in strong diatribes in the community of mathematicians and cryptographers because it was very hard to determine how to transmit a key while avoiding physically exchanging it. For example, if Alice and Bob wanted to exchange a key (before the advent of asymmetric encryption), the only trusted way to do that was to meet physically in one place. This condition caused a lot of problems with the massive adoption of telecommunication systems and the internet. The first problem was that internet communication relies on data exchange over unsafe channels. As you can easily understand, if Alice communicates with Bob through an insecure public communication channel, the private key has a severe possibility of being compromised, which is extremely dangerous for the security and privacy of communications.For this reason, this question arises: *if we use a symmetric cipher to protect our secret information, how* *can we securely exchange the secret key?*A simple answer is the following: we have to provide a *secure channel* of communication to exchange the key.Someone could then reply: *how do we provide a secure channel?*We will find the answer, or rather multiple answers, later on in this book. Even in tough military applications, such as the legendary *red line* between the leaders of the US and USSR during the Cold War, symmetric communication keys were used; nowadays, it is common to use asymmetric encryption to exchange a key. Once the key has been exchanged, the next communication session is combined with symmetric encryption to encrypt the messages transmitted.For many reasons, asymmetric encryption is a good way to exchange a key and is good for authentication and digital signatures. Computationally, symmetric encryption is better because it can work with lower bit-length keys, saving a lot of bandwidth and timing. So, in general, its algorithms work efficiently for security using keys of 256-512 bits compared to the 4,000+ bits of asymmetric RSA encryption, for example. I will explain in detail why and how that is possible later during the analysis of the algorithms in asymmetric/symmetric encryption.While in this book I will analyze many kinds of cryptographic techniques, essentially, we can group all the algorithms into two big families: symmetric and asymmetric encryption.We need some more definitions to understand cryptography well:

- **Plaintext**: In cryptography, this indicates unencrypted text, or everything that could be exposed in public. For example, **(meet you tomorrow at 10 am)** is plaintext.
- **Ciphertext**: In cryptography, this indicates the result of the text after having performed the encryption procedure. For example, **meet you tomorrow at 10 am** could become **[x549559*ehebibcm3494]** in ciphertext.

As I mentioned before, I use different brackets to identify plaintext and ciphertext. In particular, these brackets (**…**) identify plaintext, while square brackets **[…]** identify ciphertext.

Binary numbers, ASCII code, and notations

When we manipulate data with computers, it is common to use data as strings of **0** and **1** named bits. So, numbers can be converted into bits (base 2) rather than into base 10, like our numeric system. Let's just have a look at how the conversion mechanism works. For example, the number (**123**) can be written in base 10 as **1*10^2+2*10^1 +3*10^0**.Likewise, we can convert a base 10 number to a base 2 number. In this case, we use the example of the number **29**:

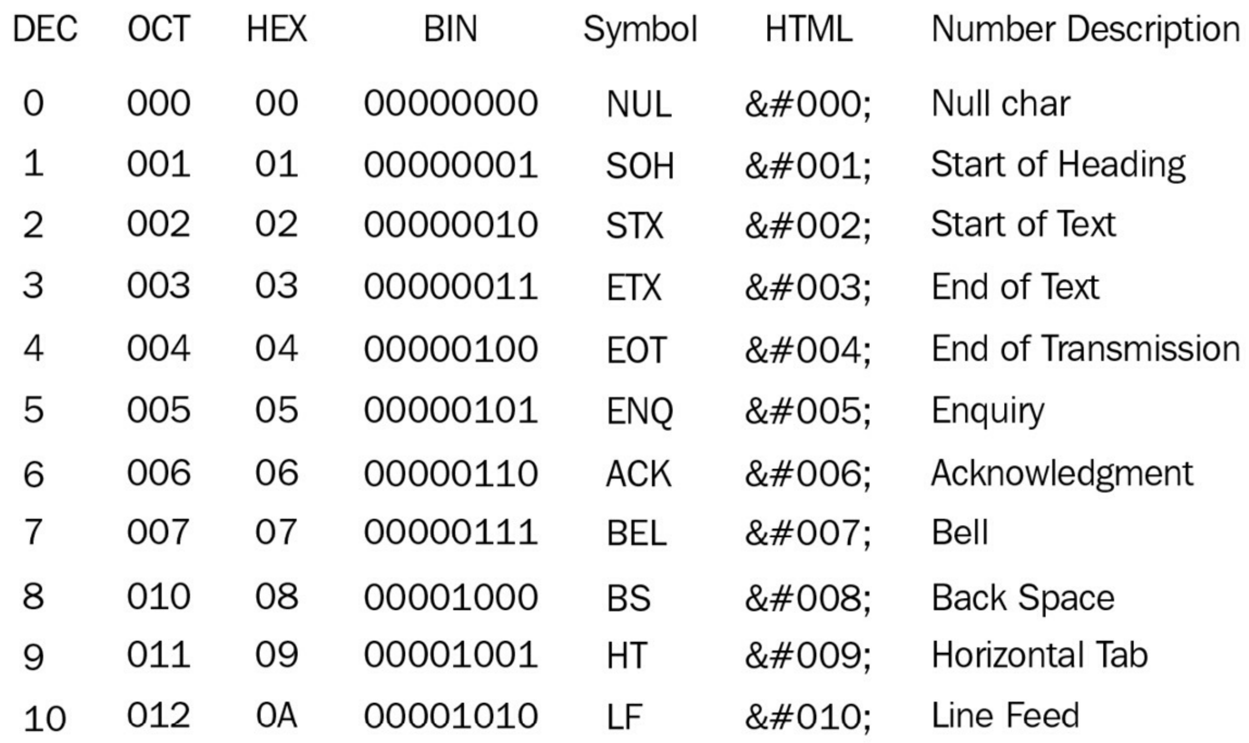


Number 29 converted into base 2 (bit)

Step	Operation	Result	Remainder	Conversion (base 2)
Step 1:	29 /2	14	1	**(11101)2**
Step 2:	14 /2	7	0	
Step 3:	7 /2	3	1	
Step 4:	3/2	1	1	
Step 5:	1 /2	0	1	

Figure 1.2 – Conversion of the number 29 into base 2 (bits)

The remainder of a division is very popular in cryptography because **modular mathematics** is based on the concept of remainders. We will go deeper into this topic in the next section, when I explain prime numbers and modular mathematics.To transform letters into a binary system to be encoded by computers, the American Standards Association invented the ASCII code in 1960.From the ASCII website, we have the following definition:*"ASCII* *stands for American Standard Code for Information Interchange. It's a 7-bit character code where every single bit represents a unique character."*The following is an example of an ASCII code table with the first 10 characters:



DEC	OCT	HEX	BIN	Symbol	HTML	Number Description
0	000	00	00000000	NUL	&#000;	Null char
1	001	01	00000001	SOH	&#001;	Start of Heading
2	002	02	00000010	STX	&#002;	Start of Text
3	003	03	00000011	ETX	&#003;	End of Text
4	004	04	00000100	EOT	&#004;	End of Transmission
5	005	05	00000101	ENQ	&#005;	Enquiry
6	006	06	00000110	ACK	&#006;	Acknowledgment
7	007	07	00000111	BEL	&#007;	Bell
8	010	08	00001000	BS	&#008;	Back Space
9	011	09	00001001	HT	&#009;	Horizontal Tab
10	012	0A	00001010	LF	&#010;	Line Feed

Figure 1.3 – The first 10 characters and symbols expressed in ASCII code

Note that I will often use in my implementations, made with the **Wolfram Mathematica** research software, the character **88** as **X** to denote the message number to encrypt. In ASCII code, the number **88** corresponds to the symbol **X**, as you can see in the following example: `88 130 58 01011000 X &#88; Uppercase X` You can go to the *Appendix* section at the end of the book to find all the notation used in this book both for the algorithms and their implementation with Mathematica code.

Fermat's Last Theorem, prime numbers, and modular mathematics

When we talk about cryptography, we have to always keep in mind that this subject is essentially related to mathematics and logic. Before I start explaining **Fermat's Last Theorem**, I want to introduce some basic notation that will be used throughout the book to prevent confusion and for a better understanding of the topic. It's important to know that some symbols, such as **=**, **≡** (equivalent), and **:=** (this last one you can find in Mathematica to compute =), will be used by me interchangeably. It's just a way to tell you that two elements correspond to each other in equal measure; it doesn't matter whether it is in a finite field (don't worry, you will become familiar with this terminology), computer science, or in regular algebra. Mathematicians may be horrified by this, but I trust your intelligence and that you will look for the substance and not for the uniformity.Another symbol, **≈** (approximate), can be used to denote similar approximative elements.You will also encounter the ^ (exponent) symbol in cases such as in a classical way to express exponentiation: ax (**a** elevated to **x**), for example. The **≠** symbol, as you should remember from high school, means **not equal** or **unequal**, which is the same as the meaning of =, that is, not equivalent.However, you will always get an explanation of the equations, so if you are not very familiar with mathematical and logical notation, you can rely on the descriptions. Anyway, I will explain each case as we come across new notation. A prime number is an integer that can only be divided by itself and 1, for example, 2, 3, 5, 7….23….67……p.Prime numbers are the cornerstones of mathematics because all other composite numbers originate from them.Now, let's see what Fermat's Last Theorem is, where it is applied, and why it is useful for us.Fermat's Last Theorem is one of the best and most beautiful theorems of classical mathematics strictly related to prime numbers. According to Wikipedia, *"In number theory, Fermat's Last Theorem (sometimes called Fermat's conjecture, especially in older texts) states that no three positive integers a, b, and c satisfy the equation a^n + b^n = c^n for any integer value of n greater than 2. The cases n = 1 and n = 2 have been known since antiquity to have infinitely many solutions."*In other words, it tells us that given the following equation, for any exponent, **n>3**, ≥ 3 there is no integer, **a**, **b**, or **c**, that verifies the sum: `a^n+b^n = c^n` Why is this theorem so important for us? Firstly, it's because Fermat's Last Theorem is strictly related to prime numbers. In fact, given the properties of primes, in order to demonstrate Fermat's Last Theorem, it's sufficient to demonstrate the following: `a^p+b^p ≠ c^p` Here, **p** is any prime number greater than 2.Fermat stated he had *a proof that was too large to fit in the margin of his notes*.Fermat himself noted in a paper that he had a beautiful demonstration

of the problem, but it has never been found.Wiles' proof is more than 200 pages long was more than 200 pages long, reduced to about 130 in the last version and is immensely difficult to understand. The proof is based on elliptic curves: these curves take a particular form when they are represented in a modular form. Wiles arrived at his conclusion after 7 years and explained his proof at a mathematicians' congress in 1994. I will explain the You will discover proof and part of the logic used in the Wile’s proof when you will read *Chapter 7, Elliptic Curves*. Right now, we just assume that to demonstrate Fermat's Last Theorem, Wiles needed to rely on the Taniyama Shimura conjecture, which states that *elliptic curves over the field of rational numbers are related to modular forms*. Again, don't worry if this seems too complicated; eventually, as we progress, it will start making sense.We will deeply analyze Fermat's Last Theorem in *Chapter 6, New Algorithms in Public/Private Key Cryptography*, when I introduce the MB09 algorithm based on Fermat's Last Theorem, among other innovative algorithms in public/private keys. Moreover, we will analyze the elliptic curves applied in cryptography in *Chapter 7, Elliptic Curves*.Fermat was obsessed with prime numbers, just like many other mathematicians; he searched for prime numbers and their properties throughout his life. He tried to attempt to find a general formula to represent all the primes in the universe, but unluckily, Fermat, just like many other mathematicians, only managed to construct a formula for some of them. The following is *Fermat's prime numbers* formula:
`2^2n + 1 for some positive integer n` If we substitute **n** with integers, we can obtain some prime numbers:
`n = 1, p = 5n = 2, p = 17n = 3, p = 65 (not prime)n = 4, p = 257`
more famous but very similar is the Mersenne prime numbers formula: `2^n - 1 for some positive integer nn = 1, p=1n = 2, p=3n =`
countless attempts to find a formula that exclusively represents all prime numbers, nobody has reached this goal as yet.**Great Internet Mersenne Prime Search** (**GIMPS**) is a research project that aims to discover the newest and biggest prime numbers with Mersenne's formula.If you explore the GIMPS website, you can discover the following: *All exponents below 53 423 543 have been tested and verified.All exponents below 92 111 363 have been tested at least once.51st Known Mersenne Prime Found!December 21, 2018 — The Great Internet Mersenne Prime Search (GIMPS) has discovered the largest known prime number, 2^82,589,933-1, having 24,862,048 digits. A computer volunteered by Patrick Laroche from Ocala, Florida made the*

*find on December 7, 2018. The new prime number, also known as M82589933, is calculated by multiplying together 82,589,933 twos and then subtracting one. It is more than one and a half million digits larger than the previous record prime number.*Besides that, GIMPS is probably the first decentralized example of how to split CPU and computer power to reach a common goal. But why all this interest in finding big primes?There are at least three answers to this question: the passion for pure research, the money – because there are several prizes for those who find big primes – and finally, because prime numbers are important for cryptography, just like oxygen is for humans. This is also the reason why there is prize money for discovering big prime numbers.You will understand that most algorithms of the next generation work with prime numbers. But how do you discover whether a number is prime?In mathematics, there is a substantial computation difference between the operation of multiplication and division. Division is a lot more computationally expensive than multiplication. This means, for instance, that if I compute **2^x**, where **x** is a huge number, it is easy to operate the power elevation but is extremely difficult to find the divisors of that number.Because of this, mathematicians such as Fermat struggled to find algorithms to make this computation easier.In the field of prime numbers, Fermat produced another very interesting theorem, known as **Fermat's Last Theorem Fermat’s Little Theorem.** Before explaining this theorem, it is time to understand what modular arithmetics is and how to compute with it.The simplest way to learn modular arithmetics is to think of a clock. When we say: *"Hey, we can meet at 1 p.m."* actually we calculate that 1 is the first hour after 12 (the clock finishes its circular *wrap*).So, we can say that we are unconsciously calculating in modulus 12 written by the notation (**mod 12**), where integers *wrap around* when reaching a certain value (in this case 12), called the modulus.Technically, the result of a calculation with a modulus consists of the remainder of the division between the number and the modulus.For example, in our clock, we have the following: `13 ≡ 1 (mod 12)` This means that **13** is *congruent* to **1** in modulus **12**. You can consider *congruent* to mean equal. In other words, we can say that the remainder of the division of **13:12** is **1**:

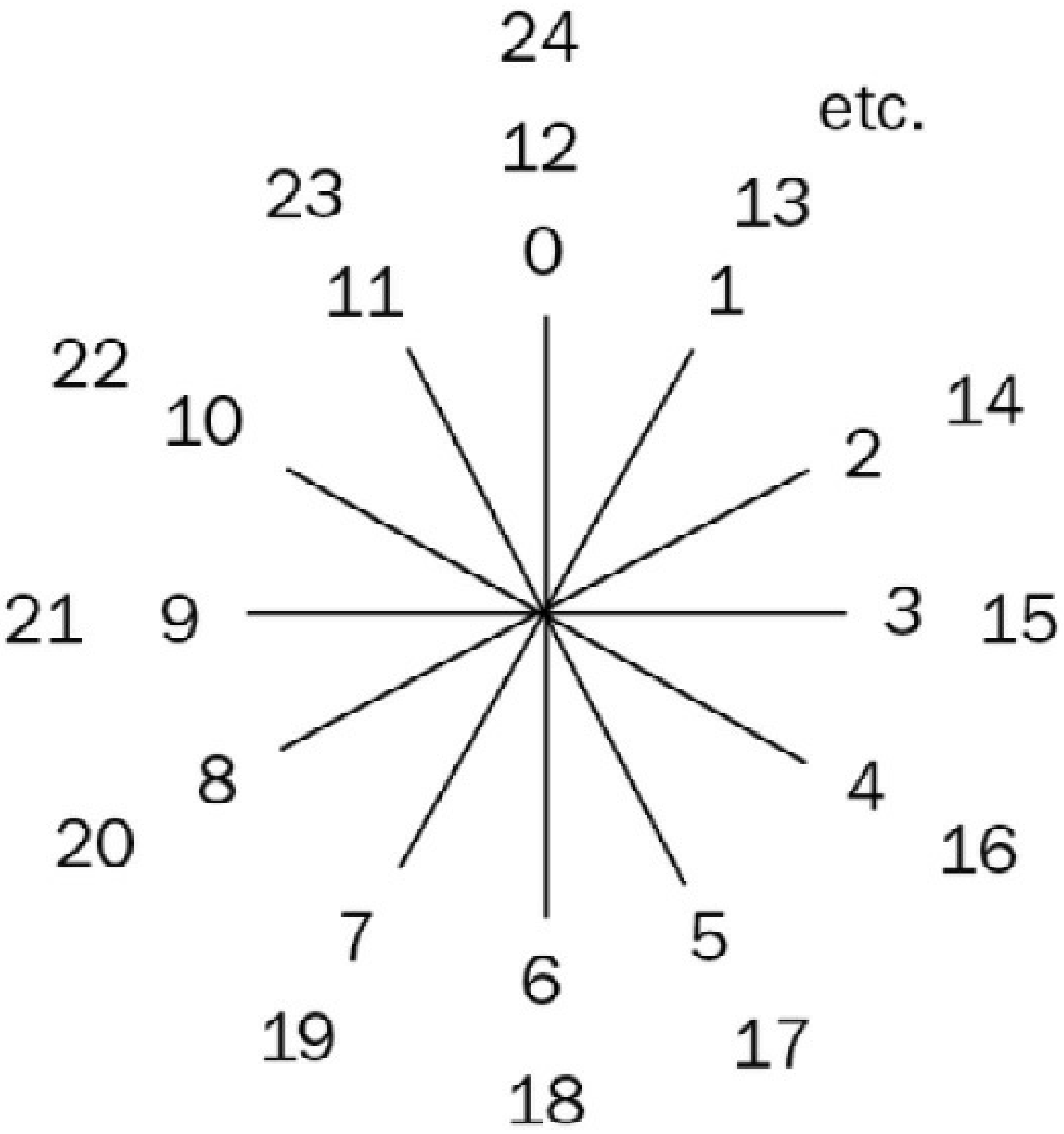


Figure 1.4 – Example of modular arithmetic with a clock

Fermat's Last Theorem Fermat’s Little Theorem states that *if* **(p)** *is a prime number, then for any integer* **(a)** *elevated to the prime number* **(p)** *we find* **(a)** *as the result of the following equation*: `a^p ≡ a (mod p)` For example, if **a = 2** and **p = 3**, then **2^3 = 2 (mod 3)**. In other terms, we find the rest of the division **8 : 3 = 2** with remainder **2**. Fermat's Last Theorem Fermat’s Little Theorem is the basis of the Fermat primality test and is one of the fundamental parts of elementary number theory.Fermat's Last Theorem Fermat’s Little Theorem states that a number, **p**, is *probably prime* in the following instance: `a^p ≡ a (mod p)` Now that we have refreshed our knowledge on the operations of bit conversion, we have seen what ASCII code looks like, and we have explored the basic notation of mathematics and logic, we can start our journey into cryptography.

A brief history and a panoramic overview of cryptographic algorithms

Nobody probably knows which cryptogram was the first to be invented. Cryptography has been used for a long time, approximately 4,000 years, and it has changed its paradigms a lot. First, it was a kind of hidden language, then cryptography was based on a transposition of letters in a mechanical fashion, then finally, mathematics and logic were used to solve complicated problems. What will the future hold? Probably, new methods will be invented to hide our secrets: quantum cryptography, for example, is already being experimented with and will come soon. I will explain new algorithms and methods throughout this book, but let me use this section to show you some interesting ciphers related to the *classical* period. Despite the computation power we have now, some of these algorithms have not yet been broken.

Rosetta Stone

One of the first extraordinary examples of cryptography was hieroglyphics. Cryptography means *hidden words* and comes from the union of two Greek words: *κρυπτός* (crypto) and *γράφω* (graphy). Among the many definitions of this word, we find the following: *converting ordinary plaintext into unintelligible text and vice versa*. So, we can include hieroglyphics in this definition, because we discovered how to *re-convert* their hidden meaning into intelligible text only after the *Rosetta Stone* was found. As you will probably remember from elementary school, the Rosetta Stone was written in three different languages: Ancient Egyptian (using hieroglyphics), Demotic, and Ancient Greek. The Rosetta Stone could only be decrypted because Ancient Greek was well known at the time:

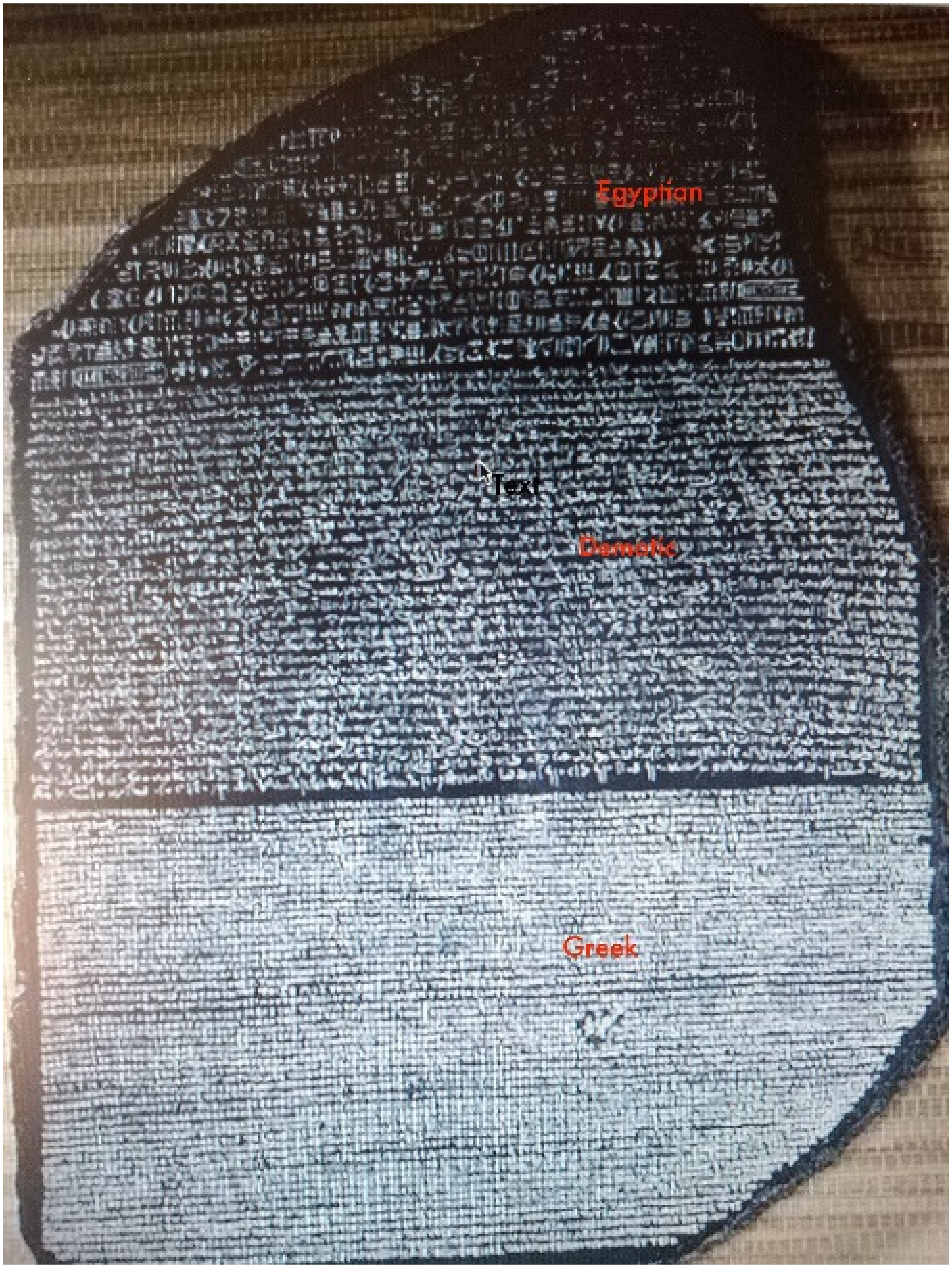

Egyptian
Text
Demotic
Greek

Figure 1.5 – Rosetta Stone with the three languages detected

Hieroglyphics were a form of communication between the people of a country. Jean-François Champollion has been recognized as the man who deciphered Rosetta Stone beginning in 1822. However, the polymath Thomas Young has been accredited by Egyptologists as the first person to publish a partially correct translation of the Rosetta Stone. We will encounter again Thomas Young in Chapter 11/12, discussing Quantum Cryptography. Young was the first to discover the effects of the dualism between waves and particles related to the photons, which is very important for Quantum mechanics. The same problem of deciphering an unknown language could occur in the future if and when we get in contact with an alien population. 20 giu 2007 A project called **SETI** (https://www.seti.org/) focuses on this: *"From microbes to alien intelligence, the SETI Institute is America's only organization wholly dedicated to searching for life in the universe."* Maybe if one day we get in contact with alien creatures, we will eventually understand their language. You can imagine that hieroglyphics (at the time) appeared as impenetrable as an alien language for someone who had never encountered this form of communication.

Caesar Cipher

Continuing our journey through history, we find that during the Roman Empire, cryptography was used to transmit messages from the generals to the commanders and to soldiers. In fact, we find the famous **Caesar Cipher**. Why is this encrypting method so famous in the history of cryptography?This is not only because it was used by Caesar, who was one of the most valorous Roman statesmen/generals, but also because this method was probably the first that implemented mathematics.This cipher is widely known as a shift cipher. The technique of shifting is very simple: just shift each letter you want to encrypt a fixed number of places in the alphabet so that the final effect will be to obtain a substitution of each letter for another one. So, for example, if I decide to shift by three letters, then *A* will become *D*, *E* becomes *H*, and so on.For example, in this case, by shifting each letter three places, implicitly we have created a secret cryptographic key of **[K=3]**:

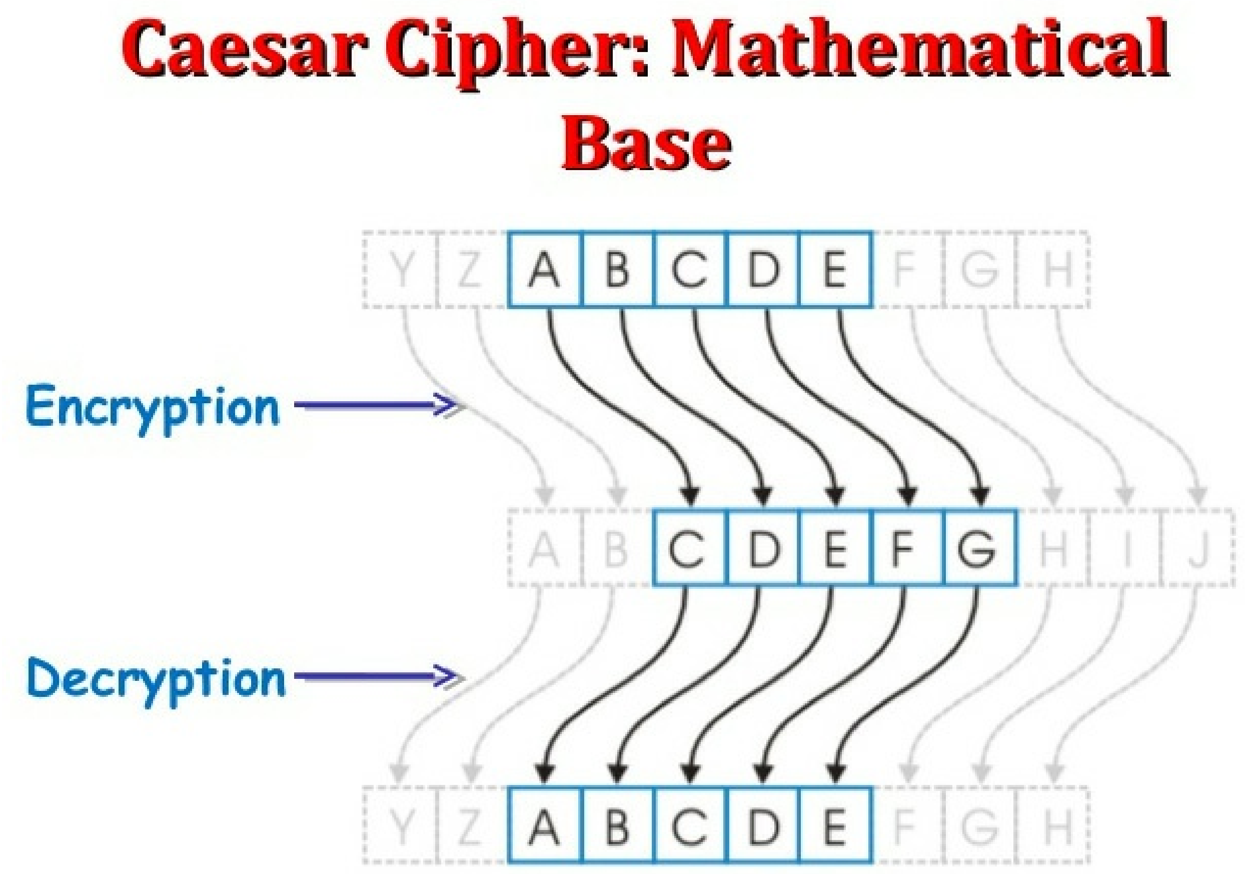


Figure 1.6 – The transposition of the letters in the Caesar Cipher during the encryption and decryption processes

It is obviously a symmetric key encryption method. In this case, the algorithm works in the following way:

- Use this key: **(+3)**.
- Message: **HELLO**.
- To encrypt: Take every letter and shift by **+3** steps.
- To decrypt: Take every letter and de-shift by **-3**.

You can see in the following figure how the process of encryption and decryption of the Caesar algorithm works using **key = +3**; as you'll notice, the word **HELLO** becomes **KHOOR** after encryption, and then it returns to **HELLO** after decryption:

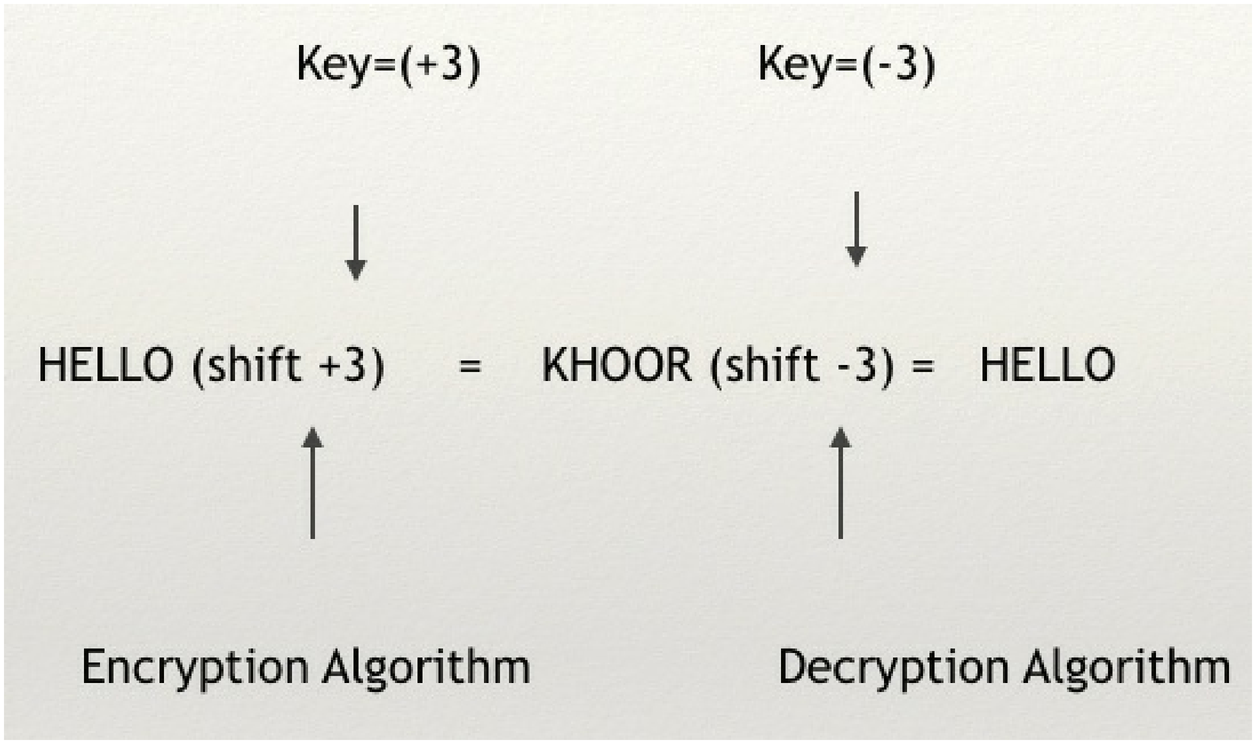


Figure 1.7 – Encryption and decryption using Caesar's algorithm

As you can imagine, the Caesar algorithm is very easy to break with a normal computer if we set a fixed key, as in the preceding example. The scheme is very simple, which, for a cryptographic algorithm, is not a problem. However, the main problem is the extreme linearity of the underlying mathematics. Using a brute-force method, that is, a test that tries all the combinations to discover the key after having guessed the algorithm used (in this case, the shift cipher), we can easily break the code. We have to check at most 25 combinations: all the letters of the English alphabet (26) minus one (that is, the same intelligible plaintext form). This is nothing compared to the billions and billions of attempts that a computer has to make in order to break a modern cryptographic algorithm.However, there is a more complex version of this algorithm that enormously increases the efficiency of the encryption.If I change the key for each letter and I use that key to substitute the letters and generate the ciphertext, then things become very interesting.Let's see what happens if we encrypt **HELLO** using a method like this:

- Write out the alphabet.
- Choose a passphrase (also known as a keyphrase) such as

[JULIUSCAESAR] and repeat it, putting each letter of the alphabet in correspondence with a character from the passphrase in the second row, as shown in the following screenshot.

- After we have defined the message to encrypt, for each character composing the message (in the first row) select the corresponding character of the keyphrase (in the second row).
- Pick up the selected corresponding characters in the second row to create the ciphertext.

Finding it a little bit complicated? Don't worry, the following example will clarify everything.Let's encrypt **HELLO** with the keyphrase **[JULIUSCAESARJULIUS…]**:

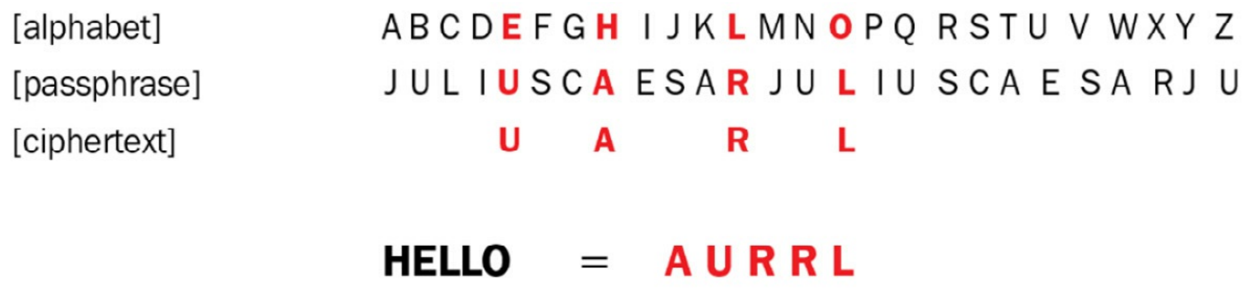

[alphabet]
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[passphrase]
J U L I U S C A E S A R J U L I U S C A E S A R J U
[ciphertext]
U A R L
HELLO = A U R R L

Figure 1.8 – Encrypting HELLO with a keyphrase becomes harder to attack

Thus, encrypting the plaintext **HELLO** using the alphabet and a key (or better, a passphrase or a keyphrase), **JULIUSCAESAR**, repeated without any spaces, we obtain the correspondent ciphertext: **AURRL**.So, **H** becomes **A**, **E** becomes **U**, **L** becomes **R** (twice), and **O** becomes **L**.Earlier, we only had to check 25 combinations to find the key in the Caesar cipher; here, things have changed a little bit, and there are (26!) possibilities to discover the key.! That means multiply *1*2*3...*26*, which results in *403,291,461,126,605,635,584,000,000*. This is undoubtedly a very big number. In fact, it is about one-third of all the atoms in the universe. Computationally, it is pretty hard to discover the key, even for a modern computer using a brute-force method.Another advantage of building a cryptogram like this is that it is easy to memorize the keyword or keyphrase and hence work out the ciphertext. But let's see a cipher that is performed with a similar technique and is used in commercial contexts.

ROT13

A modern example of an algorithm that is used on the internet is **ROT13**. Essentially, this is a simple cipher derived from the Caesar cipher with a shift of **(+13)**. Computationally, it is easy to break the Caesar cipher, but it yields an interesting effect: if we shift to the left or to the right, we will have the same result.Just like the preceding example, in ROT13, we have to select letters that correspond to the pre-selected key. Essentially, the difference here is that instead of applying a keyphrase to perform the ciphertext, we will use 13 letters from the English alphabet as the *key generator*. ROT13 takes in encryption only the letters that occur in the English alphabet and not numbers, symbols, or other characters, which are left as they are. The ROT13 function essentially encrypts the plaintext with a key determined by the first 13 letters transposed into the second 13 letters, and the inverse for the second 13 letters.Take a look at the following example to better understand the encryption scheme:

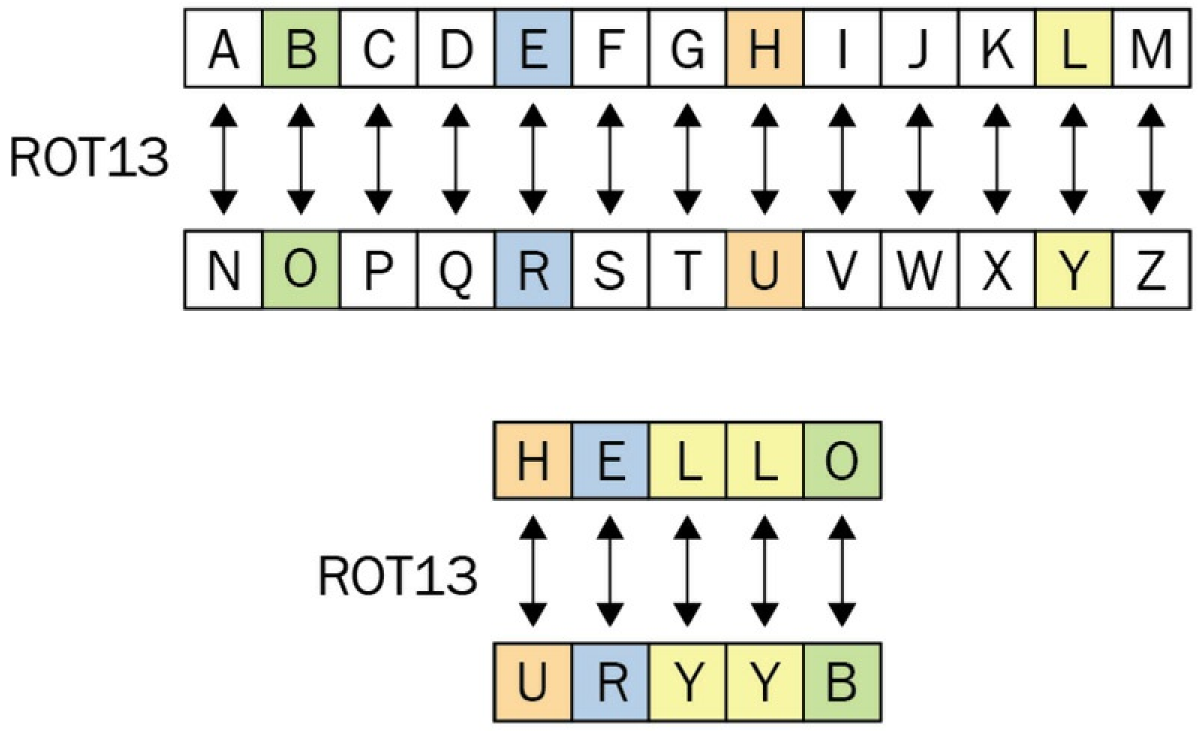


Figure 1.9 – The encryption scheme in ROT13

As you can see in the preceding diagram, **H** becomes **U**, **E** becomes **R**, **L** becomes **Y** (twice), and **O** becomes **B**: `HELLO = URYYB` The key consists of the first 13 letters of the alphabet up to **M**, which becomes **Z**, then the sequence wraps back to **N**, which becomes **A**, **O** becomes **B**, and so on to **Z**, which becomes **M**.ROT13 was used to hide potentially offensive jokes or obscure an answer in the **net.jokes** newsgroup in the early 1980s.Also, even though ROT13 is not intended to be used for a high degree of secrecy, it is still used in some cases to hide email addresses from unsophisticated spambots. ROT13 is also used for the scope of circumventing spam filters such as obscuring email content. This last function is not recommended because of the extreme vulnerability of this algorithm.However, ROT13 was used by **Netscape Communicator** – the browser organization that released https://www.mozilla.org – to store email passwords. Moreover, ROT13 is used in Windows XP to hide some registry keys, so you can understand how sometimes even big corporations can have a lack of security and privacy in communications.

The Beale cipher

Going back to the history of cryptography, I would like to show you an amazing method of encryption whose cipher has not been decrypted yet, despite the immense computational power of our modern calculators. Very often, cryptography is used to hide precious information or fascinating treasure, just as in the mysterious story that lies behind the **Beale** cipher.In order to better understand the method of encryption adopted in this cipher, I think it is interesting to know the story (or legend) of Beale and his treasure.The story involves buried treasure with a value of more than $20 million, a mysterious set of encrypted documents, Wild West cowboys, and a hotel owner who dedicated his life to struggling with the decryption of these papers. The whole story is contained in a pamphlet that was published in 1885.The story (you can find the whole version here: http://www.unmuseum.org/bealepap.htm) begins in *January 1820 in Lynchburg, Virginia* at the *Washington Hotel* where a man named *Thomas J. Beale* checked in. The owner of the hotel, *Robert Morriss,* and *Beale* became friends, and because *Mr. Morriss* was considered a trustworthy man, he received a box containing three mysterious papers covered in numbers.After countless troubles and many years of struggle, only the second of the three encrypted papers was deciphered.What exactly does Beale's cipher look like? The following content consists of three pages, containing only numbers, in an apparently random order.The first paper is as follows: `71, 194, 38, 1701, 89, 76, 11, 83, 1629, 48, 94, 63, 132,` second paper (which was decrypted) is as follows: `115, 73, 24, 807, 37, 52, 49, 17, 31, 62, 647, 22, 7, 15,` third paper is as follows: `317, 8, 92, 73, 112, 89, 67, 318, 28, 96,107, 41, 631, 78,` second cipher was successfully decrypted around 1885. Here, I will discuss the main considerations about this kind of cipher.Since the numbers in the cipher far exceed the number of letters in the alphabet, we can assume that it is not a substitution nor a transposition cipher. So, we can assume that each number represents a letter, but this letter is obtained from a word contained in an external text. A cipher following this criterion is called a **book cipher**: in the case of a book cipher, a book or any other text could be used as a key. Now, the effective key here is the method of obtaining the letters from the text.Using this system, the second cipher was decrypted by drawing on the United States Declaration of Independence. Assigning a number to each word of the referring text (the United States Declaration of Independence) and picking up the first letter of each word selected in the key (the list of the numbers, in this case, referred to the second cipher), we can extrapolate the plaintext. The extremely intelligent trick of this cipher is that the key text (the United States Declaration of Independence) is public but at the same time it was unknown to the entire world except for whom the message was intended. Only when someone holds the key (the list of the numbers) and the "key text" can they easily decrypt the message.Let's look at the process of decrypting the second cipher:

- Assign to each word of the text a number in order from the first to the last word.
- Extrapolate the first letter of each word using the numbers contained in the cipher.
- Read the plaintext.

The following is the first part of the United States Declaration of Independence (until the 115th word) showing each word with its corresponding number: `When(1) in(2) the(3) course(4) of(5) human(6) events(7) it` following numbers represent the first rows of the second cipher; as you can see, the bold words (with their corresponding numbers) correspond to the numbers we find in the ciphertext: `115, 73, 24, 807, 37, 52, 49, 17, 31, 62, 647, 22, 7, 15` following is the result of decryption using the cipher combined with the key text (the United States Declaration of Independence), picking up the first letter of each corresponding word, that is, the plaintext: `115 = instituted = I73 = hold = h24 = another = a807 (mis` so on… I haven't included the entire United States Declaration of Independence; these are only the first 115 words. But if you want, you can visit http://www.unmuseum.org/bealepap.htm and try the exercise to rebuild the entire plaintext.Here (with some missing letters) is the reconstruction of the first sentence: `I have deposited in the county of Bedford…….` If we carry on and compare the numbers with the corresponding numbers of the initial letters of the United States Declaration of Independence, the decryption will be as follows:*I have deposited in the county of Bedford, about four miles from Buford's, in an excavation or vault, six feet below the surface of the ground, the following articles, belonging jointly to the parties whose names are given in number "3," herewith:The first deposit consisted of one thousand and fourteen pounds of gold, and three thousand eight hundred and twelve pounds of silver, deposited November, 1819. The second was made December, 1821, and consisted of nineteen hundred and seven pounds of gold, and twelve hundred and eighty-eight pounds of silver; also jewels, obtained in St. Louis in exchange for silver to save transportation, and valued at $13,000.The above is securely packed in iron pots, with iron covers. The vault is roughly lined with stone, and the vessels rest on solid stone, and are covered with others. Paper number "1" describes the exact locality of the vault so that no difficulty will be had in finding it.*Many other cryptographers and cryptologists have tried to decrypt the first and third Beale ciphers in vain. Others, such as the treasure hunter *Mel Fisher*, who discovered hundreds of millions of dollars' worth of valuables under the sea, went to Bedford to search the area in order to find the treasure, without success.Maybe Beale's tale is just a legend. Or maybe it is true, but nobody will ever know where the treasure is because nobody will decrypt the first cipher. Or, the treasure will never be unearthed because someone has already found it.Anyway, what is really interesting in this story is the implementation of such a strong cipher without the help of any computers or electronic machines; it was just made with brainpower, a pen, and a sheet of paper.Paradoxically, the number of attempts required to crack the cipher go from 1 to infinity assuming that the attacker works with brute force, exploring all the texts written in the world at that moment. On top of that, what happens if a key text is not public but was written by the transmitter himself and has been kept secret? In this case, if the cryptologist doesn't have the key (so doesn't hold the key text), the likelihood of them decrypting the cipher is *zero*.The Beale cipher is also interesting because this kind of algorithm could have new applications in modern cryptography or in the future. Some of these applications could be related to methods of research for encrypted data in cloud computing.

The Vernam cipher

The **Vernam cipher** has the highest degree of security for a cipher, as it is theoretically completely secure. Since it uses a truly random key of the same length as the plaintext, it is called the **perfect cipher**. It's just a matter of entropy and randomness based on Shannon's principle of information entropy that determines an equal probability of each bit contained in the ciphertext. We will revisit this algorithm in *Chapter 8*, *Quantum Cryptography*, where we talk about quantum key distribution and the related method to encrypt the plaintext after determining the quantum key. Another interesting implementation is Hyper Crypto Satellite, which uses this algorithm to encrypt the plaintext crafted by a random key transmitted by a satellite radiocommunication and expressed as an infinite string of bits.But for now, let's go on to explore the main characteristics of this algorithm.The essential element of the algorithm is using the key only once per session. This feature makes the algorithm invulnerable to attacks against the ciphertext and even in the unlikely event that the key is stolen, it would be changed at the time of the next transmission.The method is very simple: by adding the key to the message (**mod 2**) bit by bit, we will obtain the ciphertext. We will see this method, called **XOR**, many times throughout this book, especially when we discuss symmetric encryption in *Chapter 2*, *Introduction to Symmetric Encryption*. Just remember that the key has to be of the same length as the message.A numerical example is as follows:

- **00101001** (plaintext)
- **10101100** (key): Adding each bit (**mod 2**)
- **10000101** (ciphertext)

Step 1: Transform the plaintext into a string of bits using ASCII code.**Step 2**: Generate a random key of the same length as the plaintext.**Step 3**: Encrypt by adding modulo 2 (**XOR**) of the plaintext bitwise to the key and obtain the ciphertext.**Step 4**: Decrypt by making the inverse operation of adding the ciphertext to the key and obtain the plaintext again.To make an example with numbers and letters, we will go back to **HELLO**. Let's assume that each letter corresponds to a number, starting from **0 = A**, **1 = B**, **2 = C**, **3 = D**, **4 = E** … and so on until **25 = Z**.The random key is **[DGHBC]**.The encryption will present the following transposition:

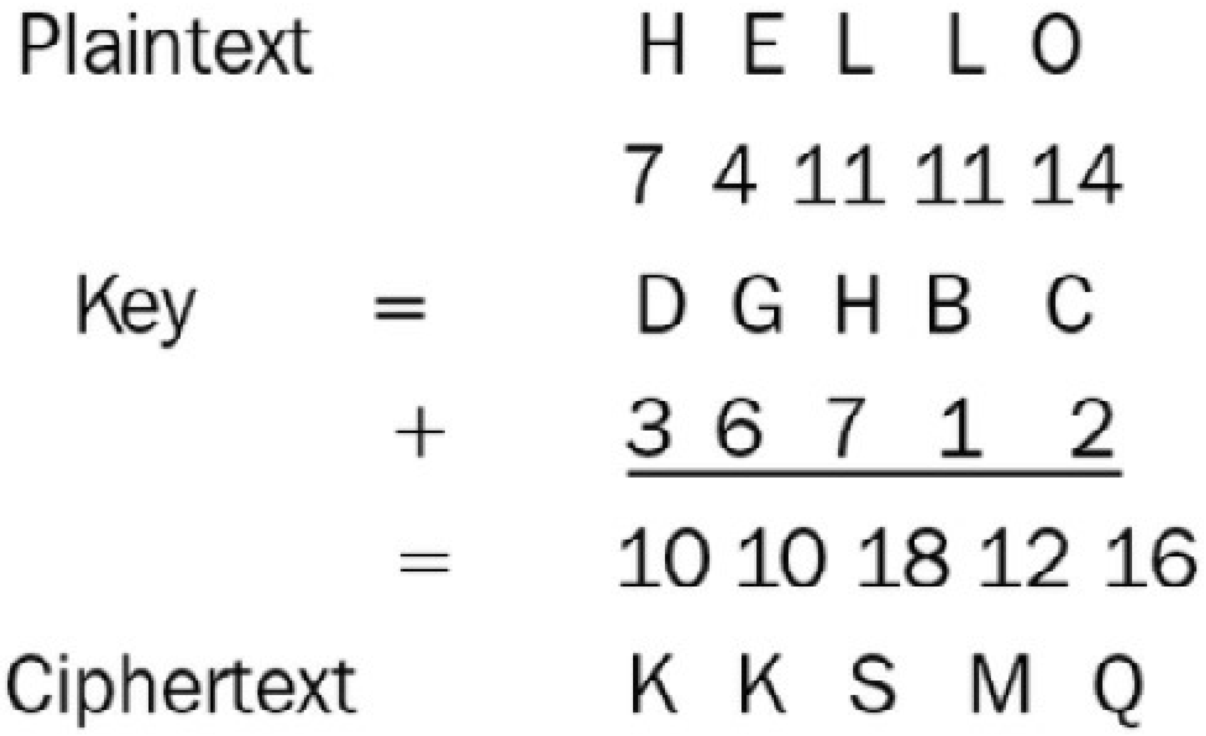


Figure 1.10 – Encryption scheme in the Vernam algorithm

So, after transposing the letters, the encryption of **[HELLO]** is **[KKSMQ]**.You can create an exercise by yourself to decrypt the **[KKSMQ]** ciphertext using the inverse process: applying **f(-K)** to the ciphertext, returning the **HELLO** plaintext. I would just like to remark that this algorithm is very strong if well implemented, following all the warnings and instructions to avoid a drastic reduction of security. One of the attacks that many algorithms suffer is well known as a ciphertext-only attack. This is successful if the attacker can deduce the plaintext, or even better the key, using the ciphertext or pieces of it. The most common techniques are frequency analysis and traffic analysis.This algorithm is not vulnerable to ciphertext-only attacks. Moreover, if a piece of a key is known, it will be possible to decipher only the piece corresponding to the related bits. The rest of the ciphertext will be difficult to decrypt if it is long enough. However, the conditions regarding the implementation of this algorithm are very restrictive in order to obtain absolute invulnerability. First of all, the generation of the key has to be completely random. Second, the key and the message have to be of the same length, and third, there is always the problem of the key transmission.This last problem affects all symmetric algorithms and is basically the problem that pushed cryptographers to invent asymmetric encryption to exchange keys between Alice and Bob (which we will see in the next chapter).The second problem concerns the length of the key: if the message is too short, for instance, the word **ten**, to indicate the time of a military attack, the attacker could also rely on their good sense or on luck. It doesn't matter if there is a random key for a short message. The message could be decrypted intuitively if the attacker knows the topic of the transmission. On the other hand, if the message is very long, we are forced to use a very long key. In this case, the key will be very expensive to produce and expensive to transmit. Moreover, considering that for every new transmission the key has to be changed, the cost of implementing this cipher for commercial purposes is very high.This is why, in general, *mono-use strings* such as this were used for military purposes during the Second World War and after. As I said before, this was the legendary algorithm used for the *red line* between Washington and Moscow to encrypt communications between the leaders of the US and the USSR during the Cold War.Finally, we will analyze the implementation of this algorithm. It could be difficult to find a way to generate and transmit a random key, even if the security of the method is very high. In the last section of this book, I will show a new method for the transmission and the implementation of keys using the Vernam cipher combined with other algorithms and methods. This new **one-time pad** (**OTP**) system, named *Hyper Crypto Satellite*, could be used for both the authentication and the encryption of messages. I will also show you the possible vulnerabilities of the system and how to generate a very random key. The method was a candidate at the **Satellite International Conference on Space**, but at the time I decided not to present it to the public.

Notes on security and computation

All the algorithms we have seen in this chapter are symmetric. The basic problem that remains unsolved is the transmission of the key. As I've already said, this problem will be overcome by the asymmetric cryptography that we will explore in the next chapter. In this section, we will analyze the computational problem related to the security of cryptographic algorithms generally speaking. Later in the book, we will focus on the security of any algorithm we will analyze.To make a similitude, we can say that in cryptography the weak link of the chain destroys the entire chain. That is the same problem as using a very strong cryptographic algorithm to protect the data but leaving its password on the computer screen. In other words, a cryptographic algorithm has to be made of a similar grade of security with respect to mathematical problems. To clarify this concept with an example: factorization and discrete logarithm problems hold similar computational characteristics for now; however, if tomorrow one of these problems were solved, then an algorithm that is based on both would be unuseful.Let's go deeper to analyze some of the principles universally recognized in cryptography. The first statement is *Cryptography has to be open source*.With the term *open source*, I am referring to the algorithm and not, obviously, to the key. In other words, we have to rely on **Kerckhoffs' principle**, which states the following: *"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.Kerckhoffs' principle applies beyond codes and ciphers to security systems in general: every secret creates a potential failure point. Secrecy, in other words, is a prime cause of brittleness—and therefore something likely to make a system prone to catastrophic collapse. Conversely, openness provides ductility."– Bruce Schneier*In practice, the algorithm that underlies the encryption code has to be known. It's not useful and is also dangerous to rely on the secrecy of the algorithm in order to exchange secret messages. The reason is that, essentially, if an algorithm has to be used by an open community (just like the internet), it is impossible to keep it secret.The second statement is *The security of an algorithm depends largely on its* *underlying mathematical problem*.As an example, RSA, one of the most famous and most widely used algorithms in the history of cryptography, is supported by the mathematical problem of factorization.Factorization is essentially the decomposition of a number into its divisors: `21 = 3 x 7` It's very easy to find the divisors of **21**, which are **3** and **7**, for small integers, but it is also well known that increasing the number of digits will exponentially increase the problem of factorization.We will deeply analyze asymmetric algorithms such as RSA in this book, and in particular, in *Chapter 3, Asymmetric Encryption,* when I will explain asymmetric encryption. But here, it is sufficient to explain why RSA is used to protect financial, intelligence, and other kinds of very sensitive secrets.The reason is that the mathematical problem underlining RSA (factorization) is still a hard problem to solve for computers of this generation. However, in this introductory section, I can't go deeper into analyzing RSA, so I will limit myself to saying that RSA suffers from not only the problem of factorization as its point of attack, but there is another equally competitive, in computational terms, problem, which is the **discrete logarithm** problem. Later in the book, we will even analyze both these hard computational problems. Now, we assume (incorrectly, as 99% of cryptographic texts do) that the pillar of security underlying RSA is factorization. In *Chapter 6, New Algorithms in Public/Private Key Cryptography*, I will show an attack on the RSA algorithm depending on a problem different from factorization. It's the similitude of the weak link of the chain explained at the beginning of this section. If something in an algorithm goes wrong, the underlying security of the algorithm fails.Anyway, let's see what happens when we attempt to break RSA relying only on the factorization problem, using brute force. In this case, just to give you an idea of the computational power required to decompose an RSA number of 250 digits, factorizing a big semi-prime number is not easy at all if we are dealing with hundreds of digits, or thousands. Just to give you a demonstration, RSA-250 is an 829-bit number composed of 250 decimal digits and is very hard to break with a computer from the current generation.This integer was factorized in *February 2020*, with a total computation time of about 2,700 core years with **Intel Xeon Gold 6130** at 2.1 GHz. Like many factorization records, this one was performed using a grid of 100 several machines and an optimization algorithm that elevated their computation.The third statement is *Practical security is always less secure than theoretical security*.For example, if we analyze the Vernam cipher, we can easily understand how the implementation of this algorithm in practice is very difficult. So, we can say that Vernam is invulnerable but only in theoretical security, not in practical security. A corollary of this assumption is this: implementing an algorithm means putting into practice its theoretical scheme and adding much more complexity to it. So, *complexity is the enemy of security*. The more complex a system is, the more points of attack can be found.Another consideration is related to the grade of security of an algorithm. We can better understand this concept by considering Shannon's theory and the concept of *perfect secrecy*. The definition given by Claude Shannon in 1949 of perfect secrecy is based on statistics and probabilities. However, about the maximum grade of security, Shannon theorized that a ciphertext maintains perfect secrecy if an attacker's knowledge of the content of a message is the same both before and after the adversary inspects the ciphertext, attacking it with unlimited resources. That is, the message gives the adversary precisely no information about the message contents.To better understand this concept, I invite you to think of different levels or grades of security, in which any of these degrees is secure but with a decreasing gradient. In other words, the highest level is the strongest and the lowest is the weakest but in the middle, there is a zone of an indefinite grade that depends on the technological computational level of the adversary.It's not important how many degrees are supposed to be secure and how many are not. I think that, essentially, we have to consider what is certainly secure and what is not, but also what can be accepted as secure in a determinate time. With that in mind, let's see the difference between perfect secrecy and secure:

- A cryptosystem could be considered to have *perfect secrecy* if it satisfies at least two conditions:
    - It cannot be broken even if the adversary has unlimited computing power.
    - It's not possible to get any information about the message, **[m]**, and the key, **[k]**, by analyzing the cryptogram, **[c]** (that is, Vernam is a theoretically perfect secrecy system but only under determinate conditions).
- A cryptogram is *secure* even if, theoretically, an adversary can break the cryptosystem (that is, if they had quantum computational power and an algorithm of factorization that runs well) but the underlying

mathematical problem is considered at that time very hard to solve. Under some conditions, ciphers can be used (such as RSA, Diffie-Hellmann, and El Gamal) because, based on empirical evidence, factorization and discrete logarithms are still hard problems to solve.

So, the concept of security is dynamic and very fuzzy. What is secure now might not be tomorrow. What will happen to RSA and all of the classical cryptography if quantum computers become effective, or a powerful algorithm is discovered tomorrow that is able to break the factorization problem? We will come back to these questions in *Chapter 8*, *Quantum Cryptography*. For now, I can say that most *classical* cryptography algorithms will be broken by the disruptive computational power of quantum computers, but we don't know yet when this will happen.Under some conditions, we will see that the **quantum exchange of the key** can be considered a **perfect secrecy system**. But it doesn't always work, so it's not currently used. Some OTP systems could now be considered highly secure (maybe semi-perfect secrecy), but everything depends on the practical implementation. Finally, remember an important rule: a weak link in the chain destroys everything.So, in conclusion, we can note the following:

- Cryptography has to be open source (the algorithms have to be known), except for the key.
- The security of an algorithm depends largely on its underlying mathematical problem.
- Complexity is the enemy of security.
- Security is a dynamic concept: perfect security is only a theoretical issue.

Summary

In this chapter, we have covered the basic definitions of cryptography; we have refreshed our knowledge of the binary system and ASCII code, and we also explored prime numbers, Fermat's equations, and modular mathematics. Then, we had an overview of classical cryptographic algorithms such as Caesar, Beale, and Vernam.Finally, in the last section, we analyzed security in a philosophical and technical way, distinguishing the grade of security in cryptography in relation to the grade of complexity.In the next chapter, we will explore symmetric encryption, where we deep dive into algorithms such as the **Data Encryption Standard** (**DES**) and **Advanced Encryption Standard** (**AES**) families, and also address some of the issues mentioned in this chapter.

2 Symmetric Encryption Algorithms

After having an overview of cryptography, it's time now to present the principal algorithms in symmetric encryption and their logic and mathematical principles.In *Chapter 1*, *Deep Diving into Cryptography*, we saw some symmetric cryptosystems such as **ROT13** and the **Vernam cipher**. Before going further into describing modern symmetric algorithms, we need an overview of the construction of block ciphers.If you recall, symmetric encryption is performed through a key that is shared between the sender and receiver and vice versa. But how do we implement symmetric algorithms that are robust (in the sense of security) and easy to perform (computationally) at the same time? Let's see how we can answer this question by comparing asymmetric with symmetric encryption.One of the main problems with asymmetric encryption is that it is not easy to perform the operations (especially the decryption), due to the high capacity of computation required to perform such algorithms at the recommended security levels. This problem implies that asymmetric encryption is not suitable for transmitting long messages, but it's better to exchange the key. Hence, by using symmetric encryption/decryption performed with the same shared key, we obtain a smoother scheme to exchange encrypted messages.In this chapter, we will learn about the following topics:

- Understanding the basics of Boolean logic

- Understanding the basics of simplified DES
- Understanding and analyzing DES, Triple DES, and DESX
- Understanding AES (Rijndael) – the actual standard in symmetric encryption
- Implementing some logical and practical attacks on symmetric algorithms

By the end of the chapter, you will have understood how to implement, manage, and attack symmetric algorithms.

Notations and operations in Boolean logic

In order to understand the mechanism of symmetric algorithms, it is necessary to go over some notations in Boolean logic and these operations on a binary system.As we have already seen in *Chapter 1*, *Deep Diving into Cryptography*, the binary system works with a set of bits of **{0,1}**. So, dealing with Boolean functions means performing logic calculations on a sequence of bits to generate an answer that could be either **TRUE** or **FALSE**.The most frequently used functions are **XOR** (exclusive **OR**), **OR** (disjunction), and **AND** (conjunction). But there are a few other notations as well that will be explained soon.A Boolean circuit aims to determine whether a variable, **(x)**, combined with another variable, **(y)**, satisfies the condition **TRUE** or **FALSE**. This problem is called the **Boolean Satisfiability Problem** (**SAT**) and it is of particular importance in computer science. SAT was the first problem to be shown as NP-complete. The question is as follows: given a certain function, does an assignment of the values **TRUE** or **FALSE** exist such that the expression results in **TRUE**?A formula of *propositional logic* is *satisfiable* if there exists an assignment that can determine that a proposition is **TRUE**. If the result is **FALSE** for all possible variable assignments, then the proposition is said to be unsatisfiable. That is of great importance in algorithm theory, such as for the implementation of search engines, and even in hardware design or electronic circuits.Let's give an example of propositional logic:

- **Premise 1**: *If the sky is clear, then it is sunny*.
- **Premise 2**: *There are no clouds in the sky*.
- **Conclusion**: *It's TRUE that it is sunny*.

As you can see in *Figure 2.1*, starting from an input and elaborating on the logic circuit with an algorithm, we obtain a conclusion of **TRUE** or **FALSE**.All these concepts will be particularly useful in further chapters of the book, especially *Chapter 5, Introduction to Zero-Knowledge Protocols*, when we talk about **zero knowledge**, and *Chapter 9*, *Crypto Search Engine*, where we talk about a search engine that works with encrypted data:

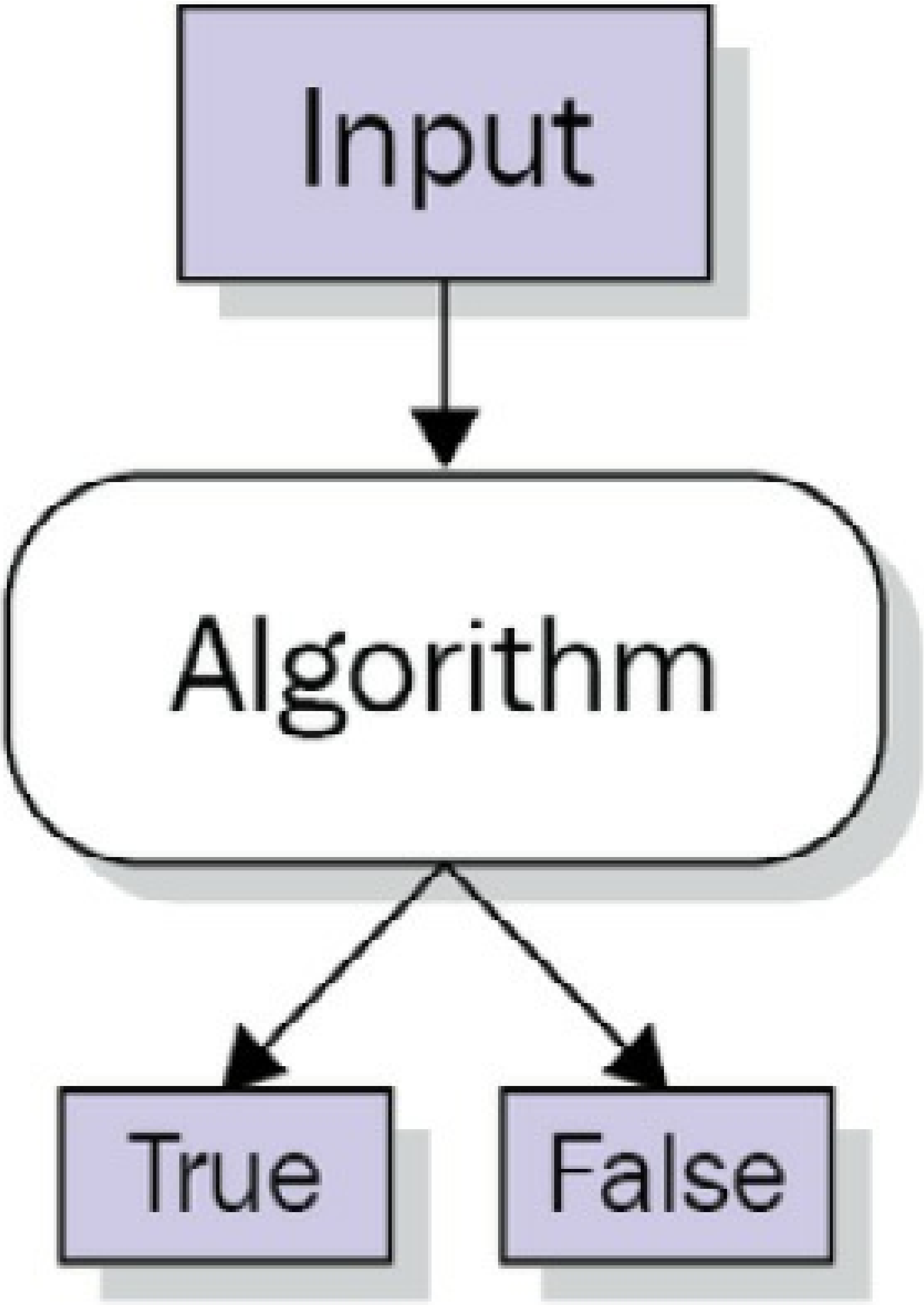

Input
Algorithm
True
False

Figure 2.1 – A Boolean circuit gives two opposite variables as output

The basic operations performed in Boolean circuits are as follows:

- **AND** (**conjunction**): Denoted with the symbol **(X^Y)**. This condition is satisfied when **X** together with **Y** is true. So, we are dealing with propositions such as **pear AND apple**, for example. If we are searching some content (let's say a database containing sentences and words), setting the **AND** operator will select all the elements containing both the words (**pear** *and* **apple**), not just one of them. Now let's explore how this operator works in mathematical mode. The **AND** operator transposed in mathematics is a multiplication of **(X * Y)**. The following is a representation of the *truth table* for all the logic combinations of the two elements. As you can see, only when **X * Y = 1** does it mean that the condition of conjunction **(X^Y)** is satisfied:

Figure 2.2 – Mathematical table for "AND"

- **OR** (**disjunction**): Denoted by the symbol **(X∨Y)**. This condition is satisfied when at least one of the elements of **X** or **Y** is true. So, we are dealing with a proposition such as **pear OR apple**. Our example of searching in a database will select all the elements containing at least one of the two words (**pear** *or* **apple**).

In the following table, you can see the **OR** operator transposed in the mathematical operation **(X+Y)**. At least one of the variables assumes the value **1**, so it satisfies the condition of disjunction **(X∨Y)**, represented by the sum of the two variables:

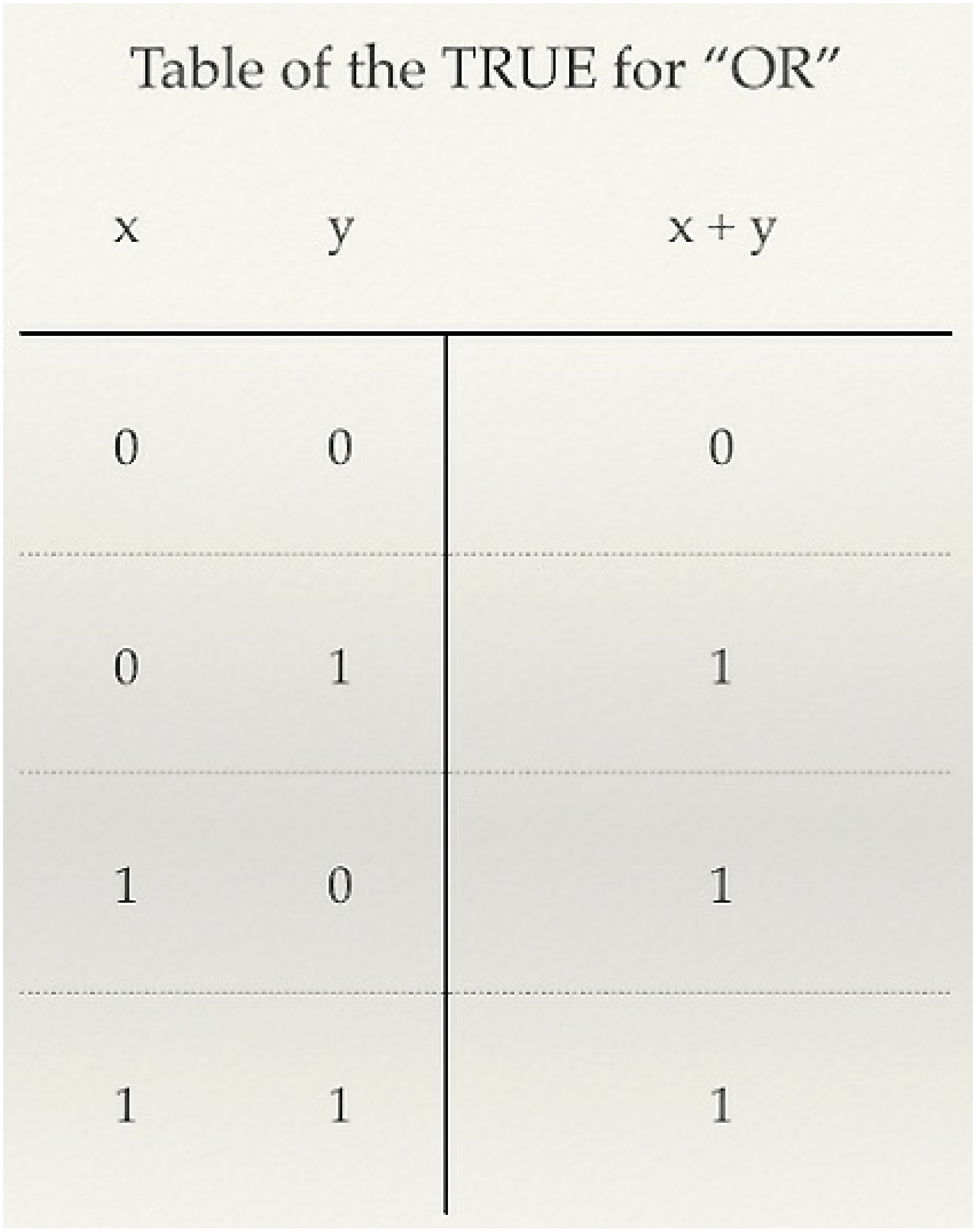


Table of the TRUE for “OR”

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.3 – Mathematical table for "OR"

Important Note

Idempotence, from *idem* + *potence* (*same* + *power*), is a property of certain operations in mathematics and computer science that denotes that they can be applied multiple times without changing the result beyond the initial application. Boolean logic has idempotence within both **AND** and **OR** gates. A logical **AND** gate with two inputs of **A** will also have an output of **A** (**1 AND 1 = 1**, **0 AND 0 = 0**). An **OR** gate has idempotence because **0 OR 0 = 0** and **1 OR 1 = 1**.

- **NOT** (**negation**): Denoted with the symbol **(¬X)**, meaning **X** excludes **Y**. So, we are dealing with propositions such as **pear NOT apple**. For example, if we search in a database, we are looking for documents containing only the first word or value (**pear**) and not for the second (**apple**). Finally, in the following table, you can see represented the **NOT** operator denoted by the symbol of negation, **(¬X)**. It is represented by a unitary operation that gets back the opposite value with respect to its input:

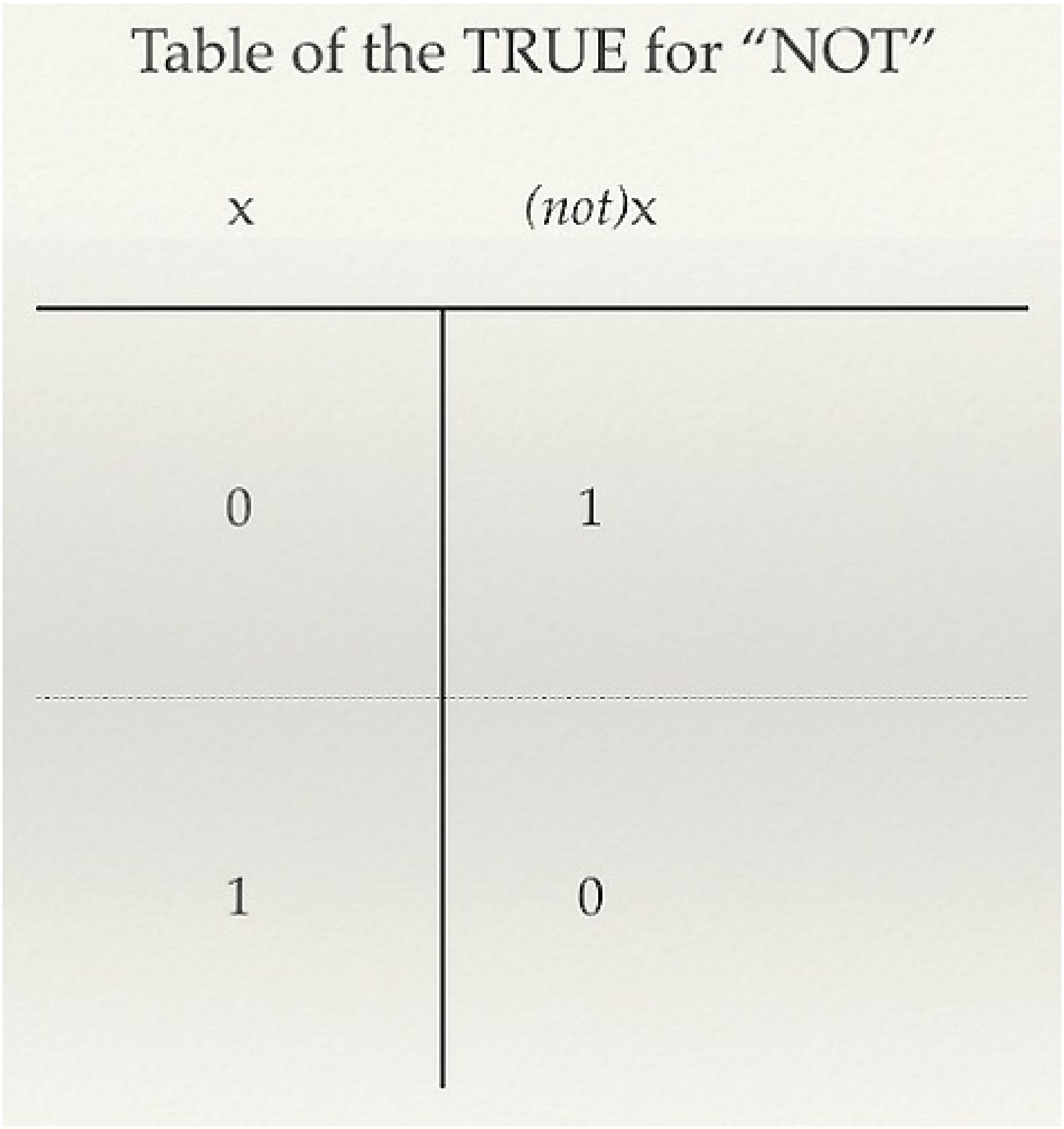


Table of the TRUE for "NOT"

x	*(not)*x
0	1
1	0

Figure 2.4 – Mathematical table for "NOT"

These basic Boolean operators, **AND**, **OR**, and **NOT**, can be represented by a Venn diagram as follows:

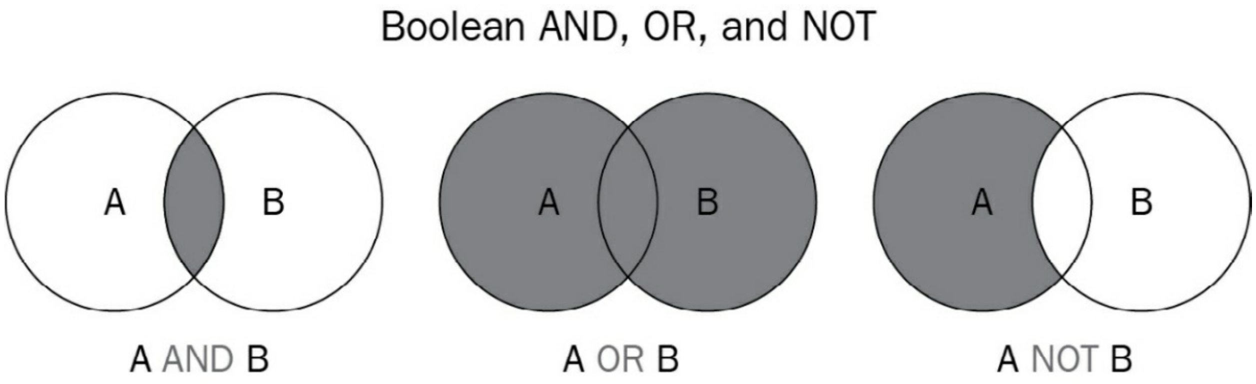
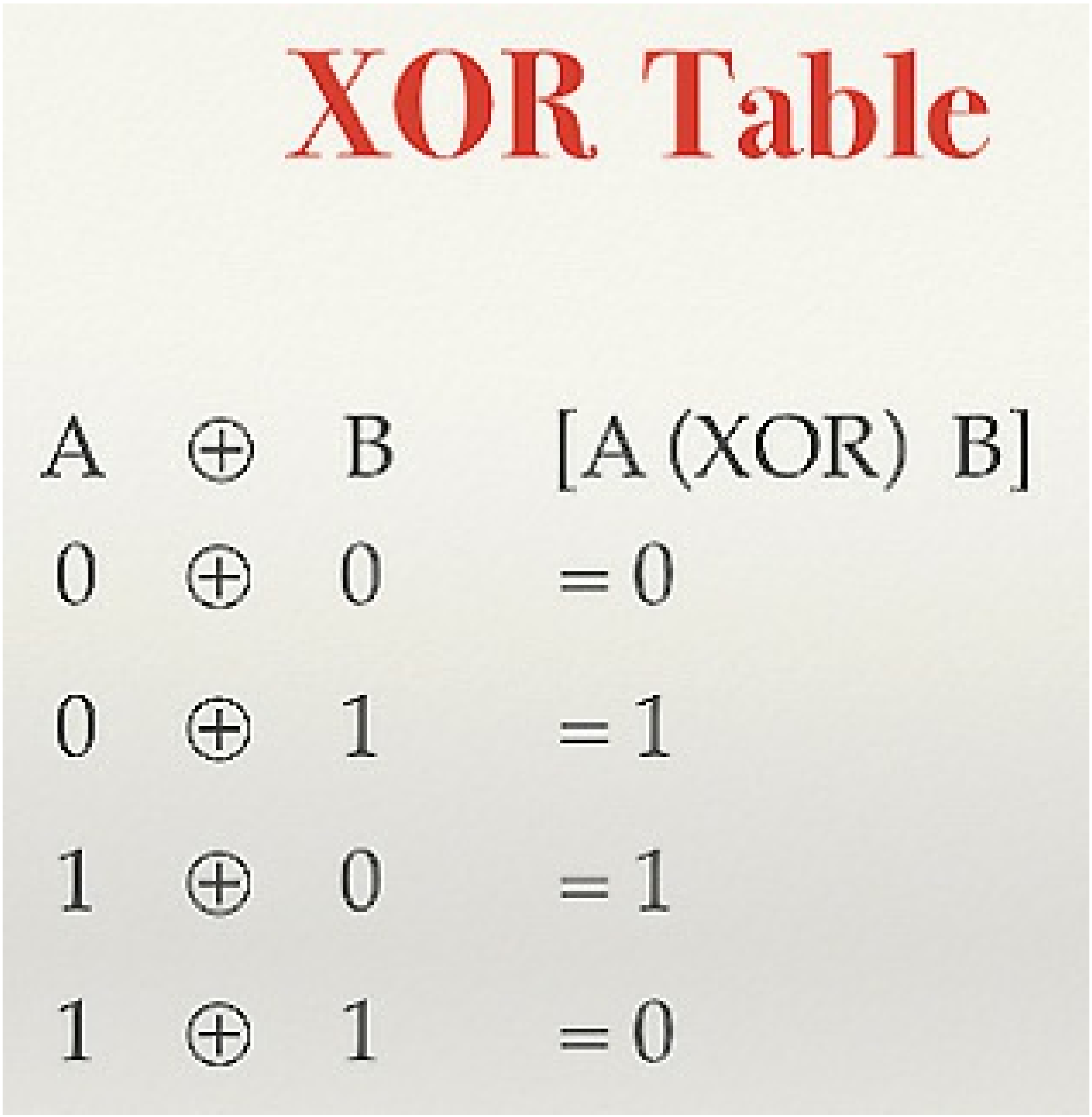


Figure 2.5 – Boolean operators represented by a Venn diagram

Besides the three basic operations just explored, there are more logic operations, including **NAND**, **NOR**, and **XOR**. All these operations are fundamental in cryptography. The **NAND** logical operator, for example, is used in **homomorphic encryption**; however, for now, we will limit ourselves to analyzing the **XOR** operator, also known as exclusive **OR**. **XOR** is also denoted by the symbol.The operation of **A B** gives back the logic value of **1** if the number of variables that assume value **1** is odd. In other words, if we consider two variables, **A** and **B**, if both are either **TRUE** or **FALSE**, then the result is **FALSE**. As we can see in the following table, when **A = 1** and **B = 1**, the result is **0** (**FALSE**). In mathematical terms, **XOR** is an addition modulo 2, which means adding combinations of **1** and **0** in **mod 2**, as you can see in the following table, is called exclusive **OR** (often abbreviated to **XOR**):



XOR Table

A	⊕	B	[A (XOR) B]
0	⊕	0	= 0
0	⊕	1	= 1
1	⊕	0	= 1
1	⊕	1	= 0

Figure 2.6 – Representing the XOR operations between 0 and 1

The **XOR** logic operator is used not only for cryptographic algorithms but also as a parity checker. If we run **XOR** in a logic circuit to check the parity bits in a word of 8 bits, it can verify whether the total number of ones in the word is a pair or not a pair.Now that we have explored the operations behind Boolean logic, it's time to analyze the first algorithm of the symmetric family: DES.

DES algorithms

The first algorithm presented in this chapter is **Data Encryption Standard** (**DES**). Its history began in 1973 when the **National Bureau of Standards** (**NBS**), which later became the **National Institute of Standards and Technology** (**NIST**), required an algorithm to adopt as a national standard. In 1974, IBM proposed **Lucifer**, a symmetric algorithm that was forwarded from NIST to the **National Security Agency** (**NSA**). After analysis and some modifications, it was renamed DES. In 1977, DES was adopted as a national standard and it was largely used in electronic commerce environments, such as in the financial field, for data encryption.Remarkable debates arose over the robustness of DES within the academic and professional community of cryptologists. The criticism derived from the short key length and the perplexity that, after a review advanced by the NSA, the algorithm could be subjected to a trapdoor, expressly injected by the NSA into DES to spy on encrypted communications. Despite the criticisms, DES was approved as a federal standard in November 1976 and was published on January 15, 1977 as **FIPS PUB 46**, authorized for use on all unclassified data. It was subsequently reaffirmed as the standard in 1983, 1988 (revised as **FIPS-46-1**), 1993 (**FIPS-46-2**), and again in 1999 (**FIPS-46-3**), the latter prescribing **Triple DES** (also known as **3DES**, covered later in the chapter). On May 26, 2002, DES was finally superseded by the **Advanced Encryption Standard** (**AES**), which I will explain later in this chapter, following a public competition. DES is a **block cipher**; this means that plaintext is first divided into blocks of 64 bits and each block is encrypted separately. The encryption process is also called the **Feistel system**, to honor *Horst Feistel*, one of the members of the team at IBM who developed Lucifer.Now that a little bit of the history of this *progenitor* of modern symmetric algorithms has been revealed, we can go further into the explanation of its logical and mathematical scheme.

Simple DES

Simple DES is nothing but a simplified version of DES. Before we delve into how DES works, let's take a look into this simplest version of DES.Just like DES, this simplified algorithm is also a block cipher, which means that plaintext is first divided into blocks. Because each block is encrypted separately, we are supposed to analyze only one block. The key, **[K]**, is made up of 9 bits and the message, **[M]**, is made up of 12 bits.The main part of the algorithm, just like in DES, is the **S-Box**, where **S** stands for **Substitution**. Here lies the true complexity and non-linear function of symmetric algorithms. The rest of the algorithm is only permutations and shifts over the bits, something that a normal computer can do automatically, so there is no reason to go crazy over it.An S-Box in this case is a 4X16 matrix consisting of 6 bits as input and 4 bits as output.We will find that the S-Box is present in all modern symmetric encryption algorithms, such as DES, Triple DES, Bluefish, Blowfish and AES.The four rows are represented by progressive 2 bits, as follows: `00011011` The 16 lines of the columns instead consist of 4 bits in this sequence: `0000 0001 0010` …… …… …… …… `1111` The matrix's boxes consist of random numbers between 0 and 15, which means they never get repeated inside the same row.In order to better understand how S-Box is implemented and how it works, here is an example: **011011**. This string of bits has two outer bits, **01**, and four middle bits, **1101**.In this case (working in the binary system, using the *N*2 notation), $(\mathbf{01})_2$ corresponds to the second row, and $(\mathbf{1101})_2$ corresponds to the 13th column. By finding the intersection of the column and the row, we obtain $(\mathbf{1001})_2$.You can see the representation in binary numbers of the S-Box matrix described here:

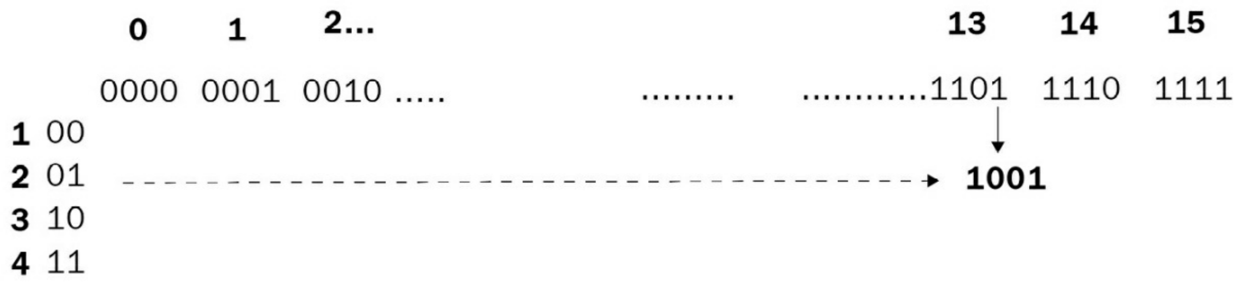
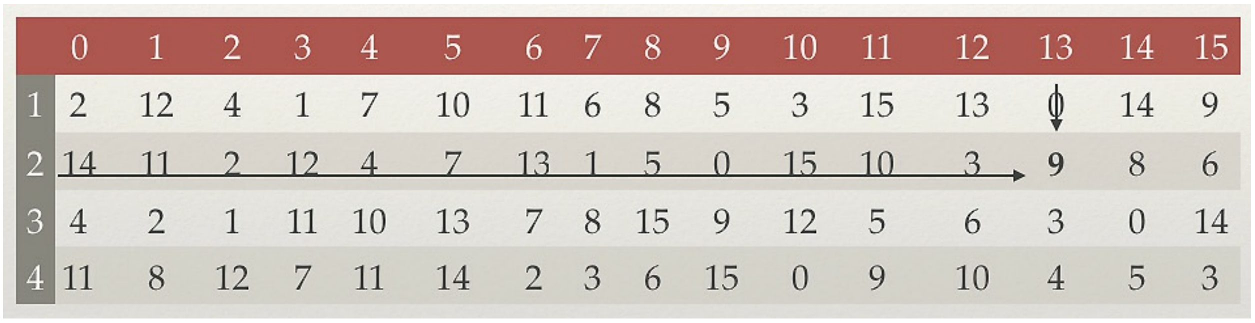


Figure 2.7 – An S-Box matrix (intersection) of 4X16 represented in binary numbers

The same matrix can be represented in decimal numbers as follows:



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

Figure 2.8 – An S-Box matrix of 4X16 represented in decimal numbers

Here, the number **9** represents the intersection between row **2** and column **13**. So, the number found crossing row **2** and column **13**, represented in a binary system as **(1001)**$_2$, corresponds to **9** in the decimal system.Now that we are clear on what S-Box is and how it is designed, we can see how the algorithm works.

Bit initialization

The message, **M**, consisting of 12 bits, is divided into two parts, **L**$_0$ and **R**$_0$, where **L**$_0$, the left half, consists of the first 6 bits and **R**$_0$, the right half, consists of the last 6 bits:

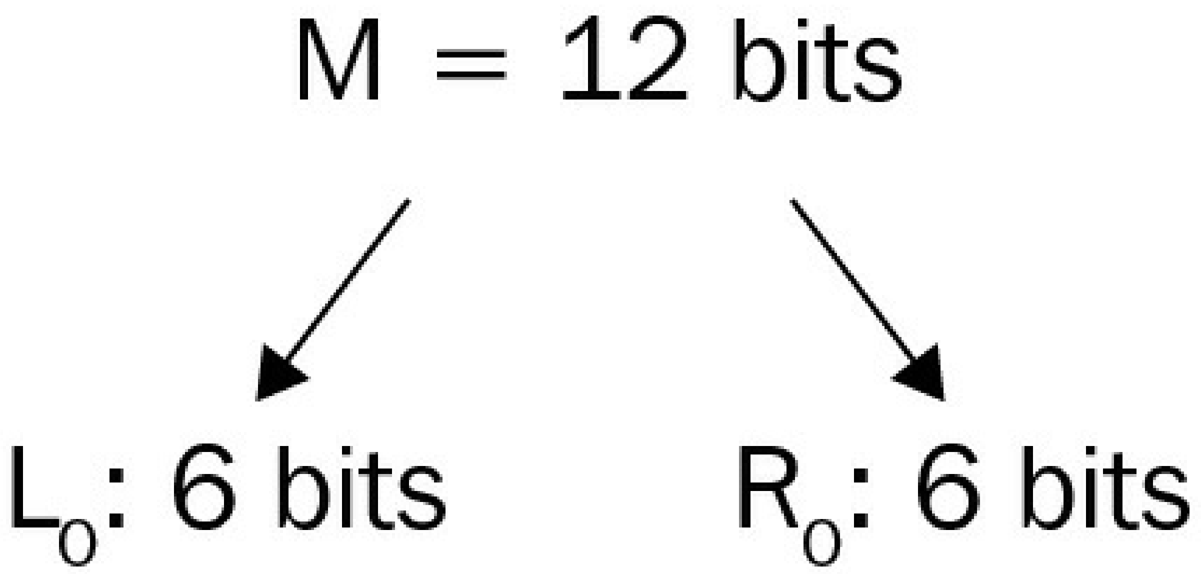


Figure 2.9 – Message (M) is split into 6 bits to the left and 6 bits to the right

Now that we have a clear concept of S-Box and bit initialization, let's proceed with the other phases of the process: bit expansion, key generation, and bit encryption.

Bit expansion

Each block of bits, the left and right parts, is expanded through a particular function that is normally called *f*. The DES algorithm uses an expansion at 8 bits (1 byte) starting from 6-bit input for each block of plaintext.Moreover, DES uses a modality of partition called **Electronic Codebook** (**ECB**) to divide the 64 bits of plaintext into 8X8 bits, for each block performing the **(E_k)** encryption function.Any *f* could be differently implemented, but just to give you an example, the first input bit becomes the first output, the third bit becomes the fourth and the sixth, and so on. Just like the following example, let's say we want to expand the 6-bit **L_0: 011001** input with an expansion function, **Exp**, following this pattern:

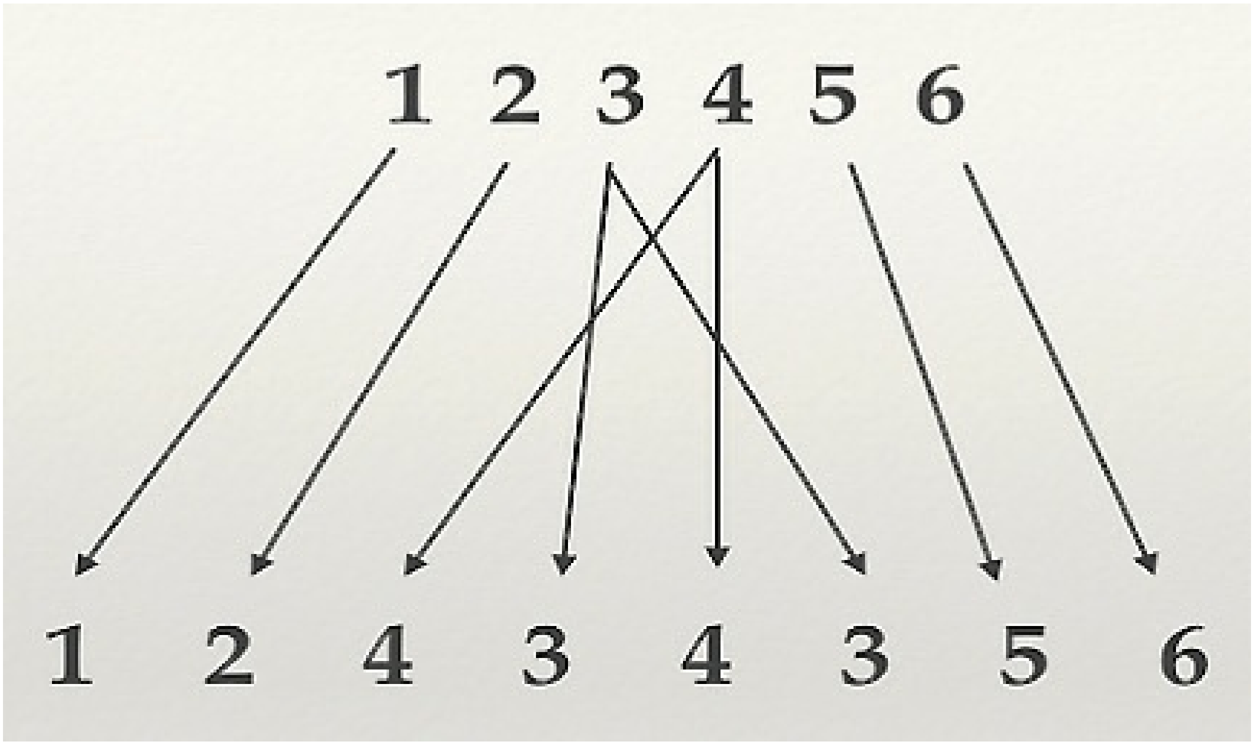


Figure 2.10 – Bit expansion function [EXP]

As you can see in the preceding figure, $\mathbf{L_0} = \mathbf{(011001)_2}$ has been expanded with **f [12434356]**. Then, $\mathbf{L_0}$**: 011001** will be expanded into $\mathbf{(01010101)_2}$, as shown in the following figure:

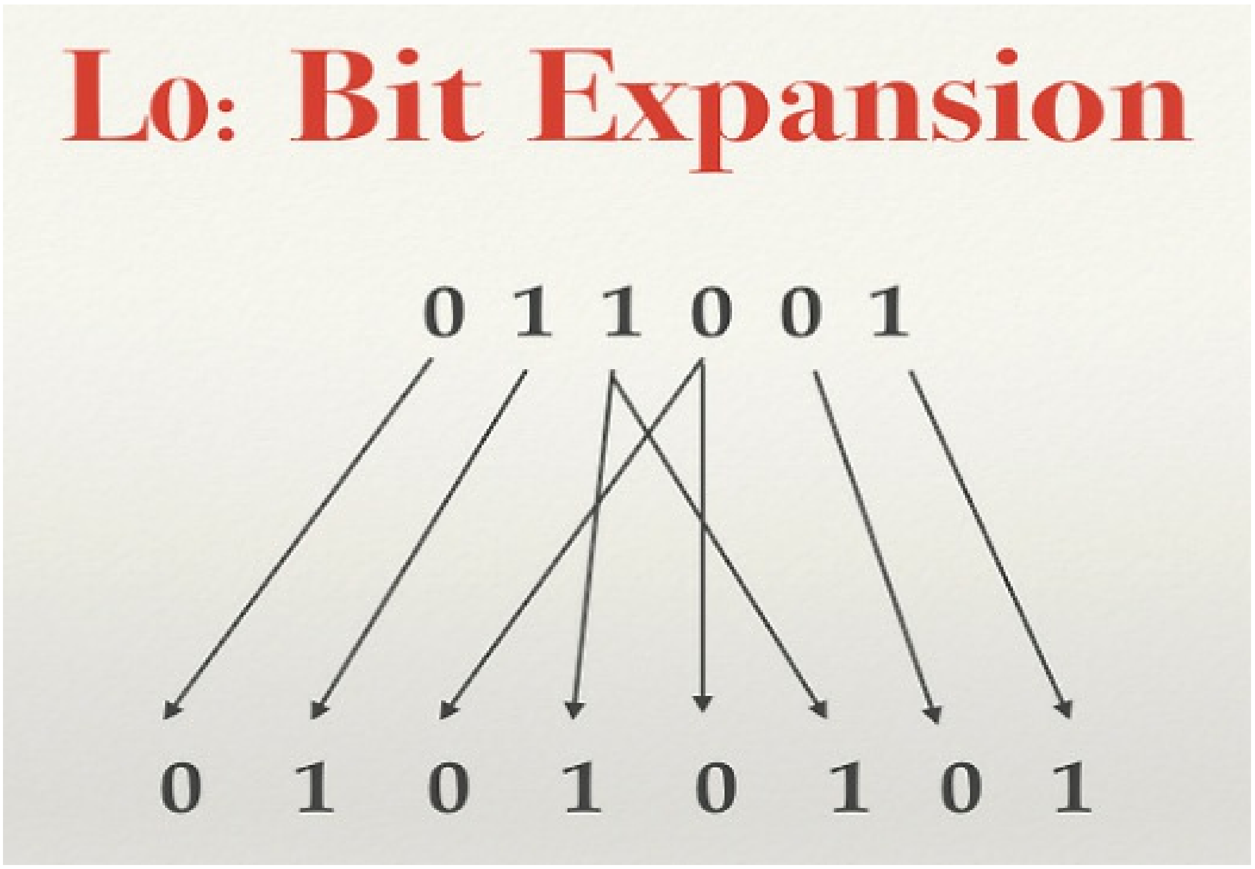


Figure 2.11 – L $_0$ (011001) $_2$ bit expansion 8 bits

Expanding the 6-bit $\mathbf{R_0}$**:** $\mathbf{(100110)_2}$ input to 8 bits with the same pattern, **f [12434356]**, in $\mathbf{R_{i-1}} = \mathbf{(100110)_2}$, we obtain the following:

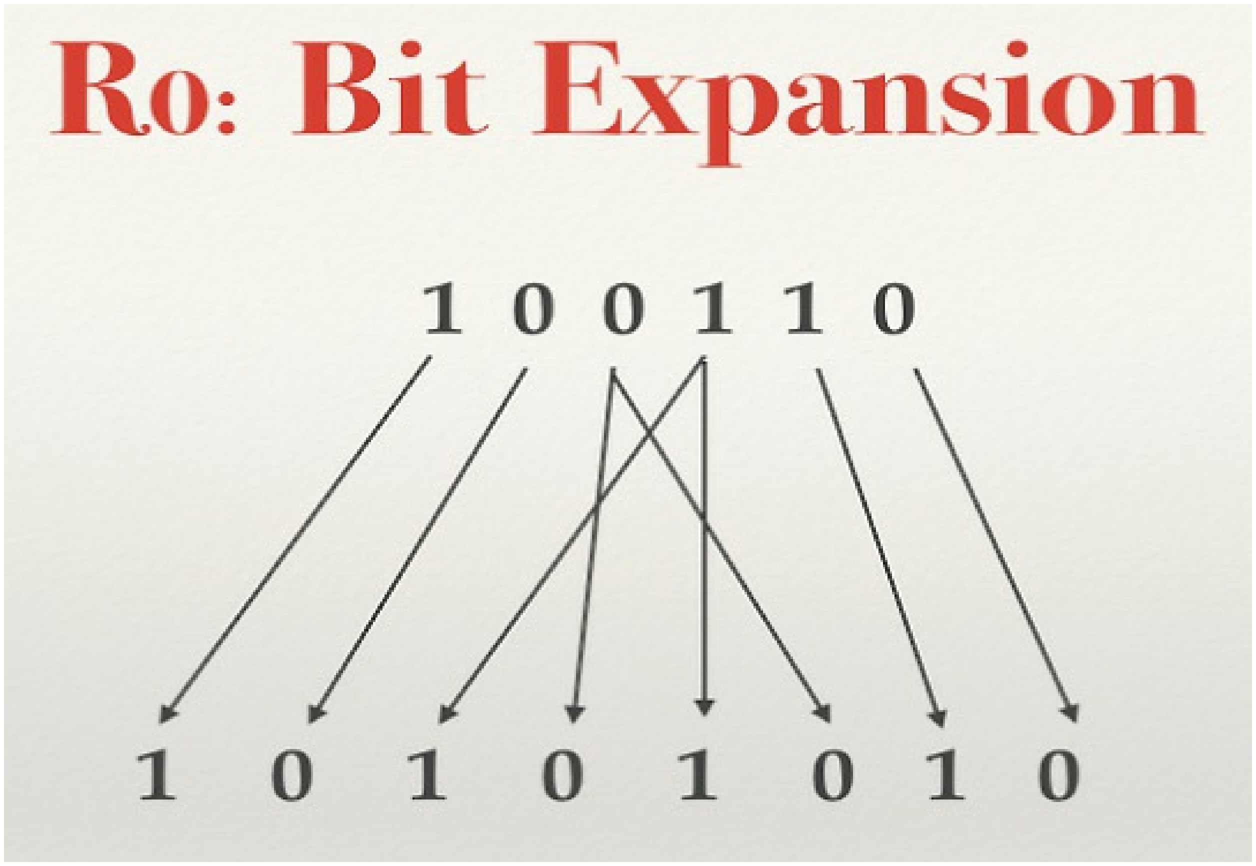


Figure 2.12 – R $_0$ (100110) $_2$ bit expansion 8 bits

So, the expansion of $\mathbf{R}_{i-1}$ will be $\mathbf{(10101010)}_2$.

Key generation

As we have already said, the master key, **[K]**, is made up of 9 bits. For each round, we have a different encryption key, **[K$_i$]**, generated by 8 bits of the master key, starting counting from the *i*th round of encryption.Let's take an example to clarify the key generation $\mathbf{K}_4$ (related to the fourth round):

- **K = 010011001** (9-bit key, the master key)
- **$\mathbf{K}_4$ = 01100101** (8 bits taken by **K**)

The following figure will help you better understand the process:

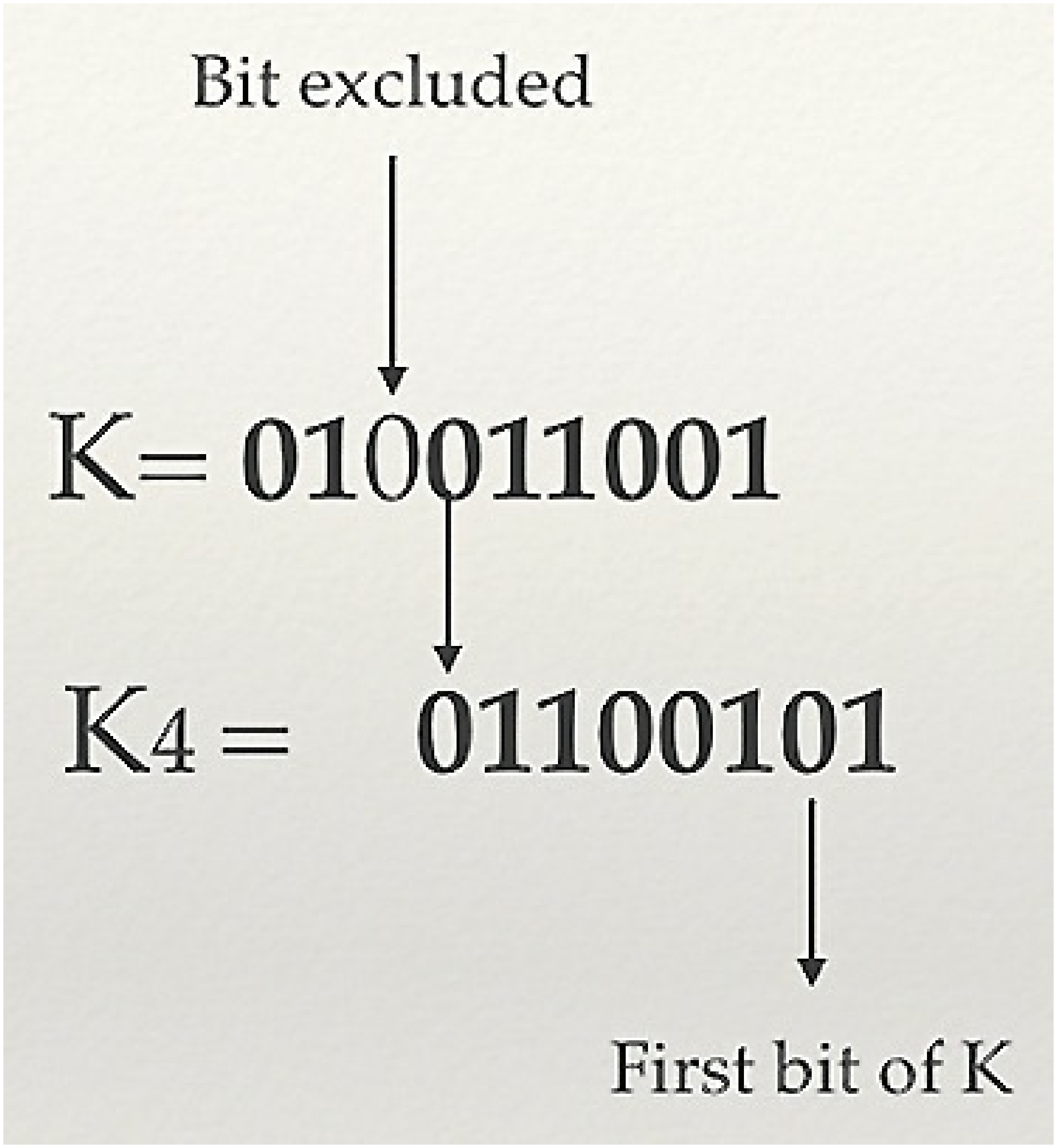
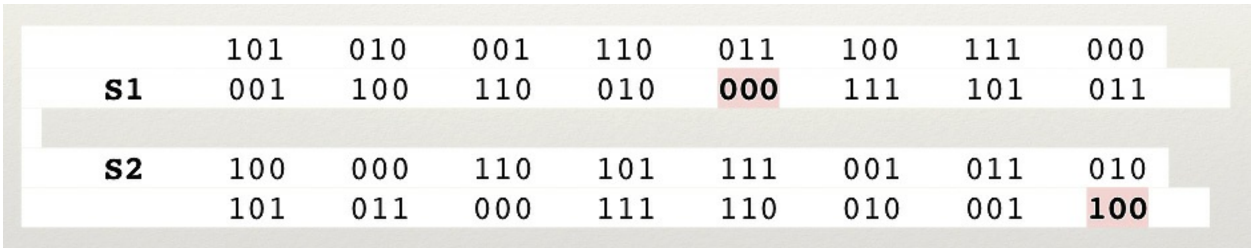


Figure 2.13 – Example of key generation

As you can see in the previous figure, we are processing the fourth round of encryption, so we start to count from the fourth bit of master key **[K]** to generate **[K4]**.

Bit encryption

To perform the bit encryption, **(E)**, we use the **XOR** function between $\mathbf{R_{i-1}}$**= (100110)2** expanded and $\mathbf{K_i}$ **= (01100101)**$_2$.I call this output **E(K$_i$)**: `Exp(Ri-1) Ki = 10101010 01100101 = E(Ki) (11001111)` At this point, we split **E(K$_i$)**, consisting of 8 bits, into two parts: a 4-bit half for the left and a 4-bit half for the right: `L(EKi)= (1100)2 R(EKi)= (1111)2.` Now, we process the 4 bits to the left and the 4 bits to the right with two S-Box 2X8 matrices consisting of 3 bits for each element. The input, as I mentioned earlier, is 4 bits: the first one is the row and the last three represent a binary number to indicate the column (the same as previously, just with fewer bits). So, **0** stands for first and **1** stands for second. Similarly, **000** stands for the first column, **001** stands for the second column, and so on until **111**.We call the two S-Boxes **S1** and **S2**. The following figure represents the elements of each one:



	101	010	001	110	011	100	111	000
S1	001	100	110	010	**000**	111	101	011
S2	100	000	110	101	111	001	011	010
	101	011	000	111	110	010	001	**100**

Figure 2.14 – Example of S-Boxes

L (E(K$_i$)) = (1100)$_2$ is processed by **S1**; so, the element of the second row, **(1)**$_2$, and the fourth column, **(100)**$_2$, is the output, here represented by the number **(000)**$_2$. **R (E(K$_i$)) = (1111)**$_2$ is processed by **S2**; so, the element of the second row, **(1)**$_2$, and the seventh column, **(111)**$_2$, is the output, here represented by the number **(100)**$_2$.Now, the last step is the concatenation of the two outputs obtained, here expressed by the notation ||, which will perform the ciphertext: `S1(L(E(Ki))) = (000)2 || S2(R(E(Ki))) = (100)2000 || 10`
following figure shows how the encryption of the first round (the right side) of the function *f* mathematically works:

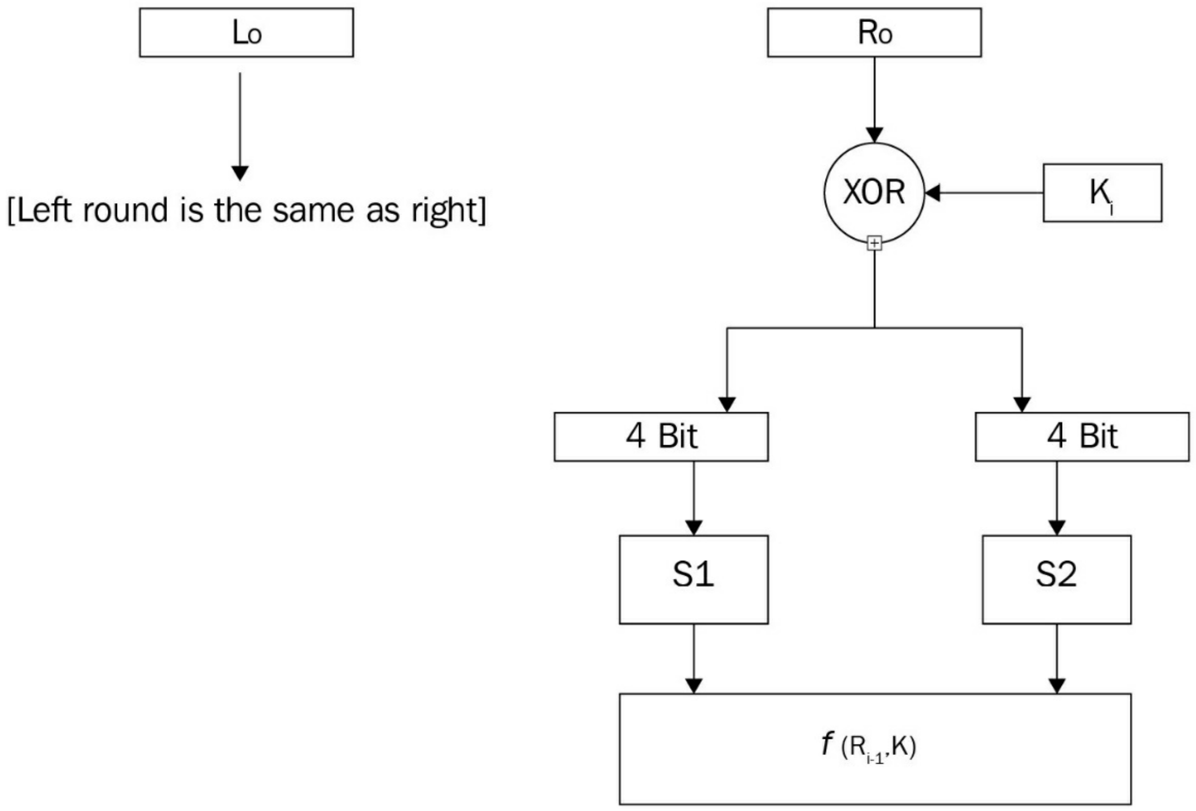


Figure 2.15 – Mathematical scheme of (simple) DES encryption at the first round (right side)

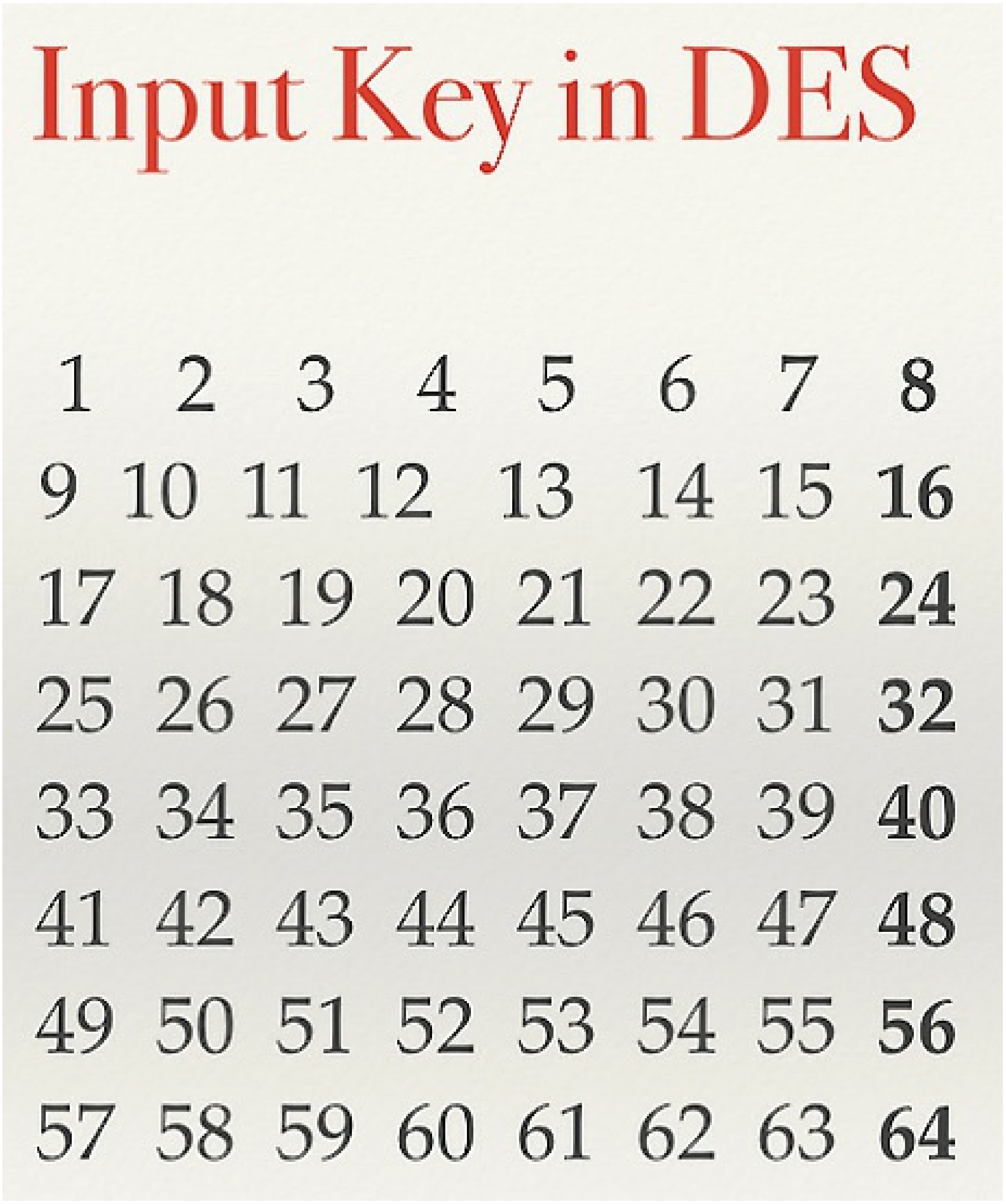
Now that we have understood how **simple DES** works and covered the basics of symmetric encryption, it will be easier to understand how the DES family of algorithms work.As you have seen, the combination of permutations, **XOR** and **shift**, is the pillar of the structure of the **Feistel system**.

DES

DES is a 16-round encryption/decryption symmetric algorithm. DES is a 64-bit cipher in every sense. The operations are performed by dividing the message, **[M]**, into 64-bit blocks. The key is also 64 bits; however, it is effectively 56 bits (plus 8 bits for parity: 8th, 16th, 24th…). This technique eventually allows us to check errors. Finally, the output, **(c)**, is 64 bits too. I would like you to focus on the DES encryption scheme of *Figure 2.15* to fully understand DES encryption.

Key generation in DES

Based on *Shannon's principle of confusion and diffusion*, DES, just like most symmetric algorithms, operates over bit scrambling to obtain these two effects.As already mentioned, the DES master key is a 64-bit key. The key's bits are enumerated from 1 to 64, where every eighth bit is ignored, as you can see in the highlighted column in the following table:



Input Key in DES

1	2	3	4	5	6	7	**8**
9	10	11	12	13	14	15	**16**
17	18	19	20	21	22	23	**24**
25	26	27	28	29	30	31	**32**
33	34	35	36	37	38	39	**40**
41	42	43	44	45	46	47	**48**
49	50	51	52	53	54	55	**56**
57	58	59	60	61	62	63	**64**

Figure 2.16 – Bits deselected in the DES master key

After the deselection of the bits, the new key is a 56-bit key. At this point, the first permutation on the 56-bit key is computed. The result of this operation is *confusion* on the bit positions; then, the key is divided into two 28-bit sub-keys called **C0** and **D0**. After this operation (always in the same line to create a bit of confusion and diffusion), it performs a circular shift process as shown in the following table:

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure 2.17 – Showing the number of key bits shifted in each round in DES

If you look at the preceding table, you can notice rounds 1, 2, 9, and 16 shift left by only 1 bit; all the other rounds shift left by 2 bits. Let's take as an example of **C0**, **D0** (the original division of the key in 28-bit left and 28-bit right), expressed in binary notation as follows:

```
C0 = 1111000011001100101010101001 D0 = 01010101011001100111100011
```

from **C0** and **D0**, **C1** and **D1** will be generated, as follows:

```
C1 = 1110000110011001010101010011 D1 = 10101010110011001111000111
```

you focus on the step of the generation of **C0 --> C1**, you can better understand how it works – it's a simple shift to the left of all the bits of **C0** with respect to **C1**:

```
C0 = 1111000011001100101010101001C1 = 111000011001100101010101001
```

the circular shift, as explained, the next step is to process a selection of 48 bits over the subset key of 56 bits. It's a simple permutation of position: just to give an example, bit number 14 moves to the first position and bit number 32 moves to the last position (48th). As you can see in the following table, some bits, just like bit number 18, are discarded in the new configuration, so you don't find them in the table. At the end of the process of bit compression, only 48 bits are selected; consequently, 8 bits are discarded:

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure 2.18 – Transformation and compression in a 48-bit subset key

In the following figure, you can see the whole process of **key generation**, which combines **Parity Drop**, **Shift Left**, and **Compression**:

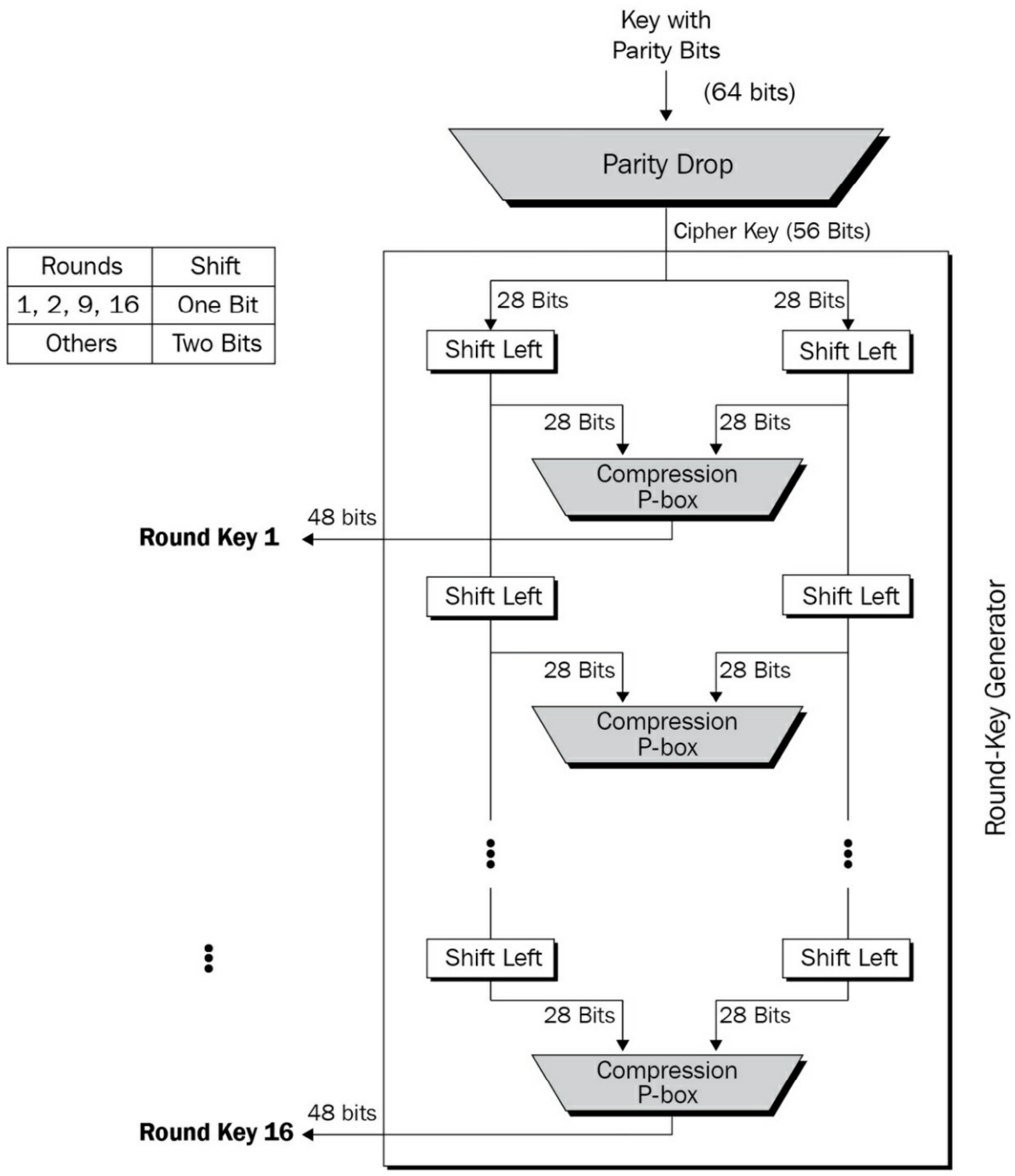


Figure 2.19 – The key generation scheme

Because of this compression/confusion/permutation technique, DES is able to determine different sub-keys, one per round of 48 bits. This makes DES difficult to crack.

Encryption

After we have generated the key, we can proceed with the encryption of the message, **[M]**.The encryption scheme of DES consists of three phases:

- **Step 1 – initial permutation**: First, the bits of the message, **[M]**, are permutated through a function that we call **Initial Permutation** (**IP**). This operation from a cryptographic point of view does not seem to augment the security of the algorithm. After the permutation, the 64 bits are divided into 32 bits in **L0** and 32 bits in **R0** just like we did in simplified DES.
- **Step 2 – rounds of encryption**: For **0 ≤ i ≤ 16**, the following operations are executed:

`Li = Ri-1` Compute **Ri = Li-1 f(Ri-1, Ki)**.Here, **Ki** is a string of 48 bits obtained from the key **K** (round key *j*) and **f** is a function of expansion similar to the **f** described earlier for simple DES.Basically, for **i = 1** (the first round), we have the following: `L1 = R0R1 = L0s f(R0, K1)`

- **Step 3 – final permutation**: The last part of the algorithm at the 16th round (the last one) consists of the following:
    - Exchanging the left part, **L16**, with the right part, **R16**, in order to obtain **(R16, L16)**
    - Applying the inverse, **INV**, of the IP to obtain the ciphertext, **c**, where **c = INV(IP (R16, L16))**

The following figure is a representation of an intelligible scheme of DES encryption:

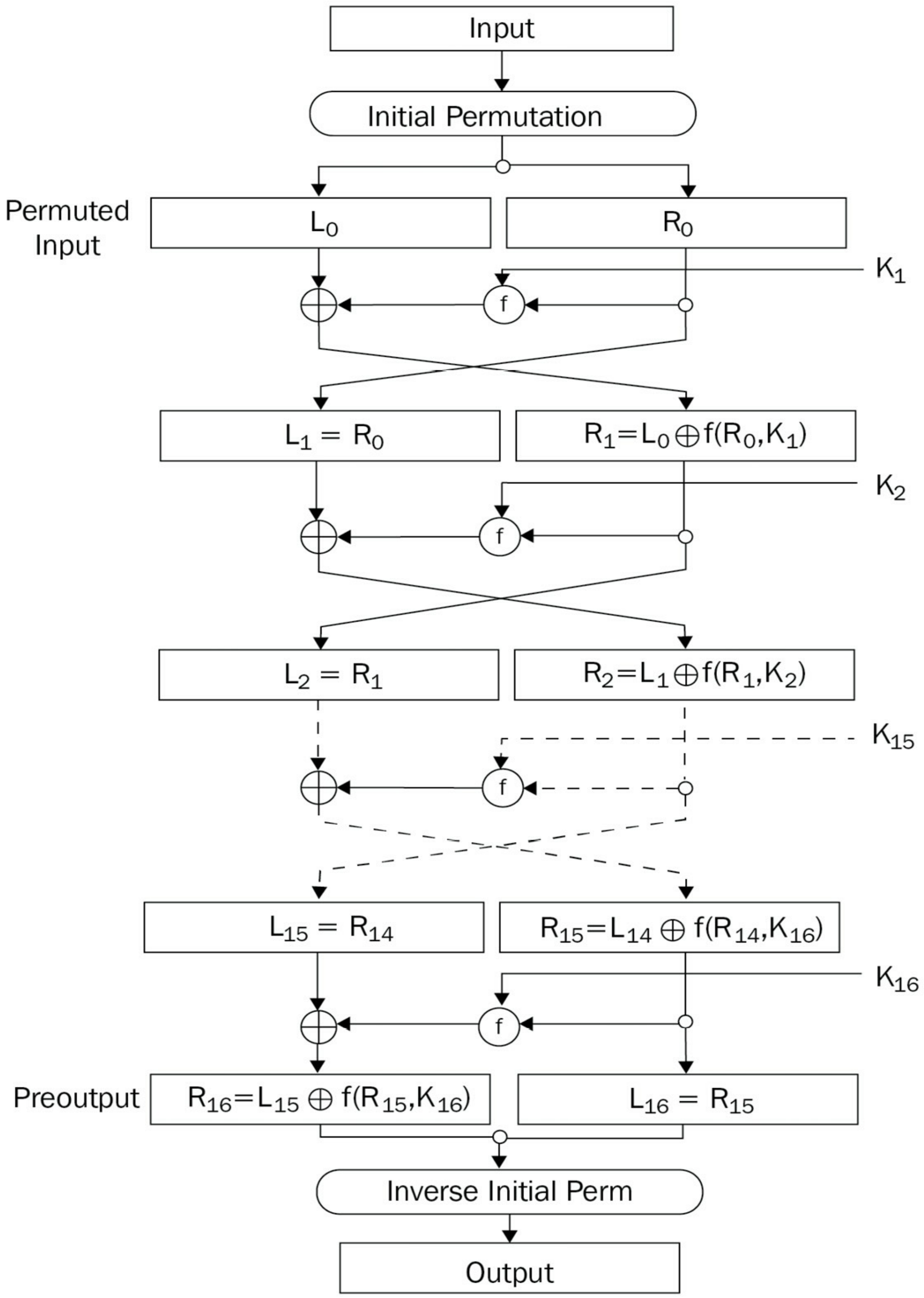

Input
Initial Permutation
Permuted Input
L_0
R_0
K_1
f
$L_1 = R_0$
$R_1 = L_0 \oplus f(R_0, K_1)$
K_2
f
$L_2 = R_1$
$R_2 = L_1 \oplus f(R_1, K_2)$
K_{15}
f
$L_{15} = R_{14}$
$R_{15} = L_{14} \oplus f(R_{14}, K_{16})$
K_{16}
f
Preoutput
$R_{16} = L_{15} \oplus f(R_{15}, K_{16})$
$L_{16} = R_{15}$
Inverse Initial Perm
Output

Figure 2.20 – DES encryption

To summarize the encryption stage in DES, we performed a complex process for key generation, where a selection of 48-bit subsection keys on a master key of 64 bits was made. There are consequently three steps: IP, rounds of encryption, and final permutation.Now that we have analyzed the encryption process, we can move on to DES decryption analysis.

Decryption

DES decryption is very easy to understand. Indeed, to get back the plaintext, we perform an inverse process of encryption.The decryption is performed in exactly the same manner as encryption, but by inverting the order of the keys ($\mathbf{K}_1\ldots\mathbf{K}_{16}$) so that it becomes ($\mathbf{K}_{16}\ldots\mathbf{K}_1$).In the following figure, you see the decryption process in a flow chart scheme:

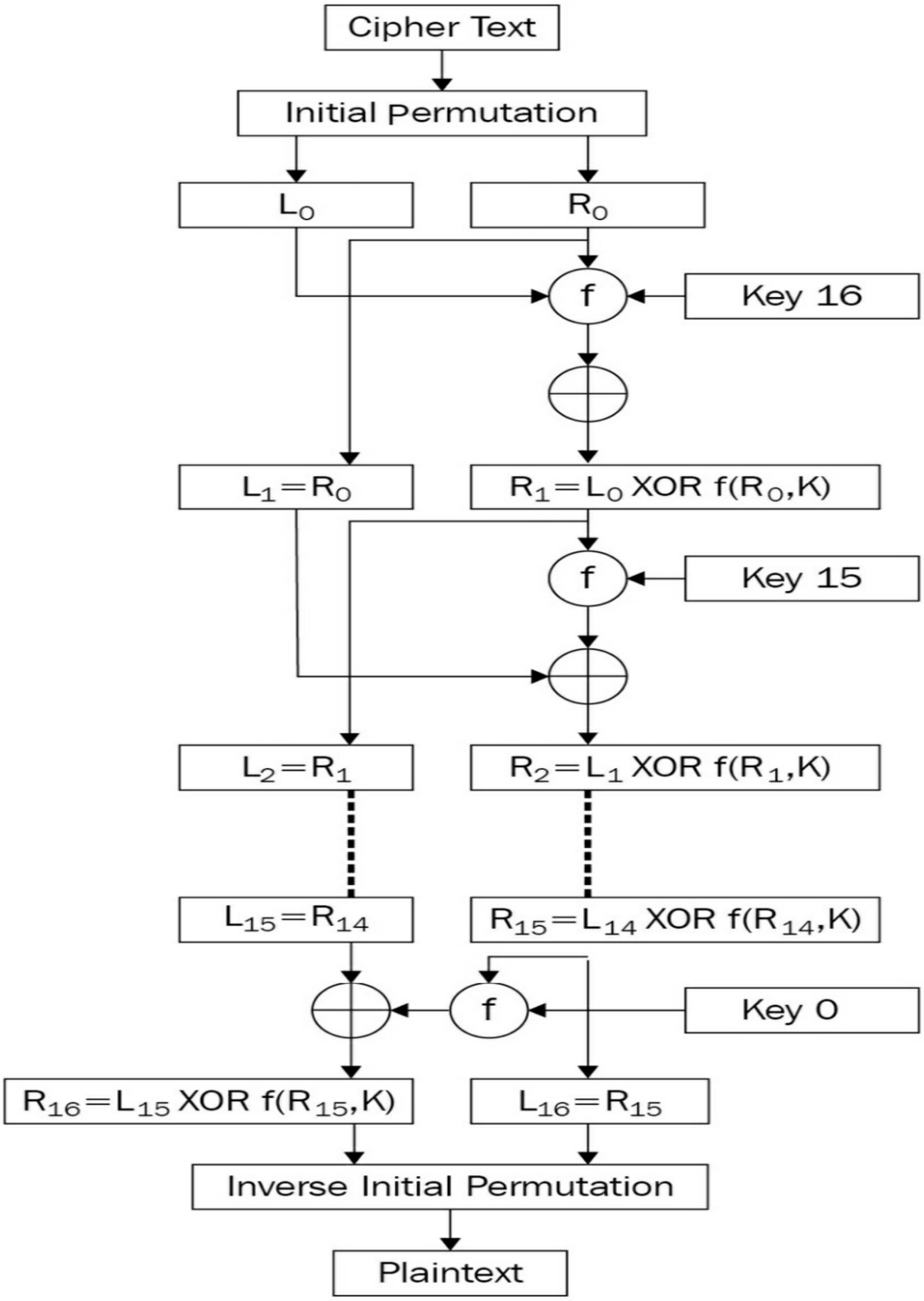

Cipher Text
Initial Permutation
L_0
R_0
f
Key 16
$L_1 = R_0$
$R_1 = L_0$ XOR $f(R_0, K)$
f
Key 15
$L_2 = R_1$
$R_2 = L_1$ XOR $f(R_1, K)$
$L_{15} = R_{14}$
$R_{15} = L_{14}$ XOR $f(R_{14}, K)$
f
Key 0
$R_{16} = L_{15}$ XOR $f(R_{15}, K)$
$L_{16} = R_{15}$
Inverse Initial Permutation
Plaintext

Figure 2.21 – DES decryption process

So, to describe the decryption process: you take the ciphertext and operate the first IP on it, then **XOR L0** (left part) with **R0 = f(R0, K16)**.Then you keep on going like that for each round, make a final permutation, and end up finding the plaintext.Now that we have arrived at the end of the DES algorithm process, let's go ahead with the analysis of the algorithm and its vulnerabilities.

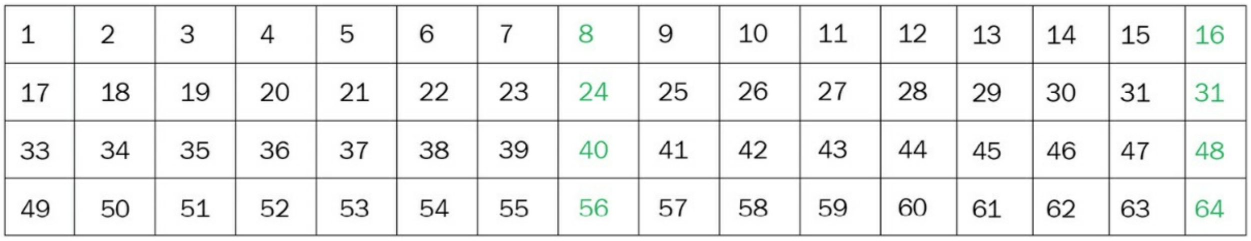
Analysis of the DES algorithm

Going a little bit more into the details of the algorithm, we can discover some interesting things.One of the most interesting steps of DES is the **XOR** operation performed between the sub-keys (**K1**, **K2**…**K16**) and the half part of the message, **[M]**, at each round.In this step, we find the S-Box: the function *f* previously described in simplified DES.As we saw earlier, the S-Box is a particular matrix in DES that consists of 4 rows and 16 columns (S-Box 4X16) fixed by the NSA:

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

Figure 2.22 – S-Box matrix in DES

Take a look at the specifics of the S-Box in the fifth 5th round of DES: `2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0 14, 914, 11, 2` you might notice, the numbers included are between **0** and **(R_{i-1}) 16-1 = 15**.So, 48 bits of input will give exactly 48 bits of output after the **XOR** operation is performed with **$[K_i]$**.Moreover, if you observe carefully, in the 14th column, all the numbers are very low: **0**, **9**, **3**, and **4**. *Would this combination pose a problem for security?* You will be perplexed if I tell you that it will not be an issue to play with little numbers inside an S-Box.Another question that may come to you spontaneously might be *Why is the key only 56 bits and not 64 bits?* Because, as I already mentioned, the other 8 bits are used for pairing. Actually, the initial master key is 64 bits in length, so every eighth bit of the key is discarded. The final result is a 56-bit key, as you see in the following figure:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	31
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure 2.23 – Bit discarded in the DES key generation algorithm

There is one more concern that could arise from the method of encryption adopted in DES. After the IP, as you can see, the bits are encrypted only on the right side through the **$f(R_{i-1}, K_i)$** function. You might ask whether this is less secure than encrypting all the bits. If you analyze the scheme properly, you will notice that at each round, the bits are exchanged from left to right, then encrypted, and vice versa. This technique is like a wrap that allows all the bits to be encrypted, not just the right part as it would seem at first glance.Looking back at the initial and final permutation functions, you may ask: when making an initial and final inverse permutation, isn't the final result neutral? As I already mentioned, there isn't any cryptographic sense in performing a permutation of bits like that. The reason is that bit insertion into hardware in the 70s was much more complicated than it is now. To complete the discussion, I can say that the entire process adopted in DES for substitution, permutation, E-expansion, and bit-shifting generates *confusion* and *diffusion*. At the beginning of this section, I already quoted this concept when I mentioned the security cipher principle identified by *Claude Shannon* in his *A Mathematical Theory of Communication*.

Violation of DES

The history of the attacks performed to crack DES since its creation is rich in anecdotes. In 1975, among the academic community, skepticism against the robustness of the key length with respect to the 56-bit keys started to arise. Many articles have been published; one very interesting prediction of *Whitney Diffie* and *Martin Hellman* (the same pair from the *Diffie and Hellman exchange of the key* seen in *Chapter 3*, *Asymmetric Encryption*) was that a computer worth $20 million (in 1977) could be built to break DES in only 1 day.More than 20 years later, in 1998, the **Electronic Frontier Foundation** (**EFF**) developed a dedicated computer called **DES Cracker** to break DES. The EFF spent a little less than $250,000 and employed 37,050 units embedded into 26 electronic boards. After 56 hours, the supercomputer gained the decryption of the plaintext message object of a challenge as there was a $10,000 reward for the decryption of the ciphertext. The method adopted was a simple *brute-force* method to analyze all the possible combinations of bits given by the **2^56** (about 72 quadrillion) possible keys. The EFF was able to crack DES using hardware that incorporated 1,500 microchips working at 40 MHz, in 4.5 days of running time. Imagine, only 1 microchip would take 38 years to explore the entire set of keys.At this point, the authorities decided to replace the algorithm with a new symmetric key algorithm, and here came AES. But before exploring AES, let's analyze some possible attacks on DES. I will present next some possible attacks on DES, taking into consideration that most of these methods are used to attack most symmetric algorithms. Some of the attacks are specific to blocking ciphers, while others are valid for streaming ciphers too. The difference is that in a stream cipher, 1 byte is encrypted at a time, while in a block cipher, ~128 bits are encrypted at a time (block):

- **Brute-force attack**: This basic method of attack can be performed for any known cipher, meaning trying all the possibilities to find the key. If you recall, the key length of DES is a 56-bit key, which means an attacker would evaluate all the 72,057,594,037,927,936 possibilities (**2^56**). But, as we need to try less than all the sets of keys, because statistically, as proved by Mitsuru Matsui, with (2^47) known-plaintexts it is possible to break 16-round DES. This is not a computation to be taken lightly, but despite this, DES has been a breakable algorithm since the early 90s. For your reference if interested you can find the Mitsuru Matsui's paper here: https://link.springer.com/content/pdf/10.1007/3-540-48285-7_33.pdf

- **Linear cryptanalysis**: This is essentially a statistical method of attack based on known plaintext. It doesn't guarantee success every time, but it does work most of the time. The idea is to start from the known input (plaintext) and arrive at determining the key of encryption and, consequently, all the other outputs generated by that key.
- **Differential cryptanalysis**: This method is technical and requires observing some vulnerabilities inside DES (similar to other symmetric algorithms). This attack method attempts to discover the plaintext or the key, starting from a chosen plaintext. Unlike linear cryptanalysis, which starts from improbable known plaintext, the attacker operates knowing chosen plaintext.

Last but not least, a vulnerability of DES is called *weak keys*: these keys are simply not able to perform any encryption. This is very dangerous because if applied, you get back plaintext. These keys are well known in cryptography and have to be avoided. That happens when the sequence of the 16th key (during the key generation) produces all 16 identical keys.Let's see an example of this problem:

- **A sequence of bits all equal to 0000000000000000** or **1111111111111111**
- **A sequence of alternate bits**, **0101010101010101** or **1010101010101010**

In all four cases, it turns out that the encryption is auto-reversible, or in other words, if you perform two encryptions on the same ciphertext, you will obtain the original plaintext.

Triple DES

As I mentioned previously, one of the main weaknesses found in DES was the key length of 56 bits. So, to amplify the volume of keys and to extend their life, a new version of DES was proposed in the form of **Triple DES**.The logic behind 3DES is the same as DES; the difference is that here we run the algorithm three times with three different keys.The following figure shows a scheme proposed to better understand 3DES:

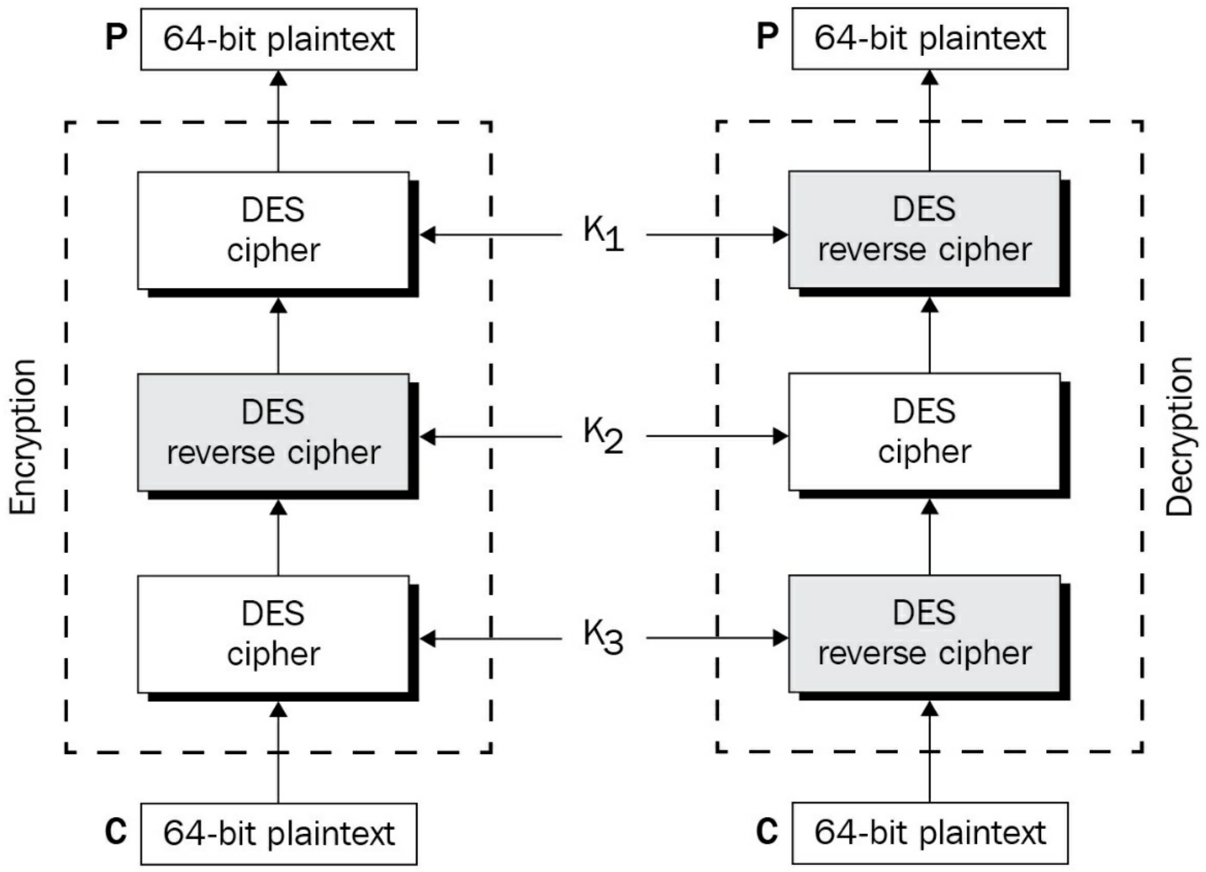


Figure 2.24 – Triple DES encryption/decryption scheme

Let's see how the encryption and decryption stages work in DES, based on the scheme illustrated in the preceding figure.Encryption in 3DES works as follows:

- Encrypt the plaintext blocks using single DES with the **[K_1]** key.
- Now, decrypt the output of *step 1* using single DES with the **[K_2]** key.
- Finally, encrypt the output of *step 2* using single DES with the **[K_3]** key.

The output of *step 3* is the ciphertext **(C)**.**Decryption in 3DES**The decryption of ciphertext is a reverse process. The user first decrypts using **[K_3]**, then decrypts with **[K_2]**, and finally decrypts with **[K_1]**.

DESX

The last algorithm of the DES family is **DESX**. This is a reinforcement of DES's key proposed by *Ronald Rivest* (the same coauthor of RSA).Given that DES encryption/decryption remains the same as earlier, there are three chosen keys: **[K_1]**, **[K_2]**, and **[K_3]**.The following encryption is performed: `C = [K3] EK1 ([K2] [M])` First, we have to perform the encryption **(E_K)**, making an **XOR** between **[K_2]** and the message, **[M]**. Then, we apply DES, encrypting with **[K_1]** 56 bits. Finally, we add the outputs **E_{K1}**, **XOR**, and **[K_3]**. This method allows us to increase the virtual key to be **64 + 56 + 64 = 184** bits, instead of the normal 56 bits:

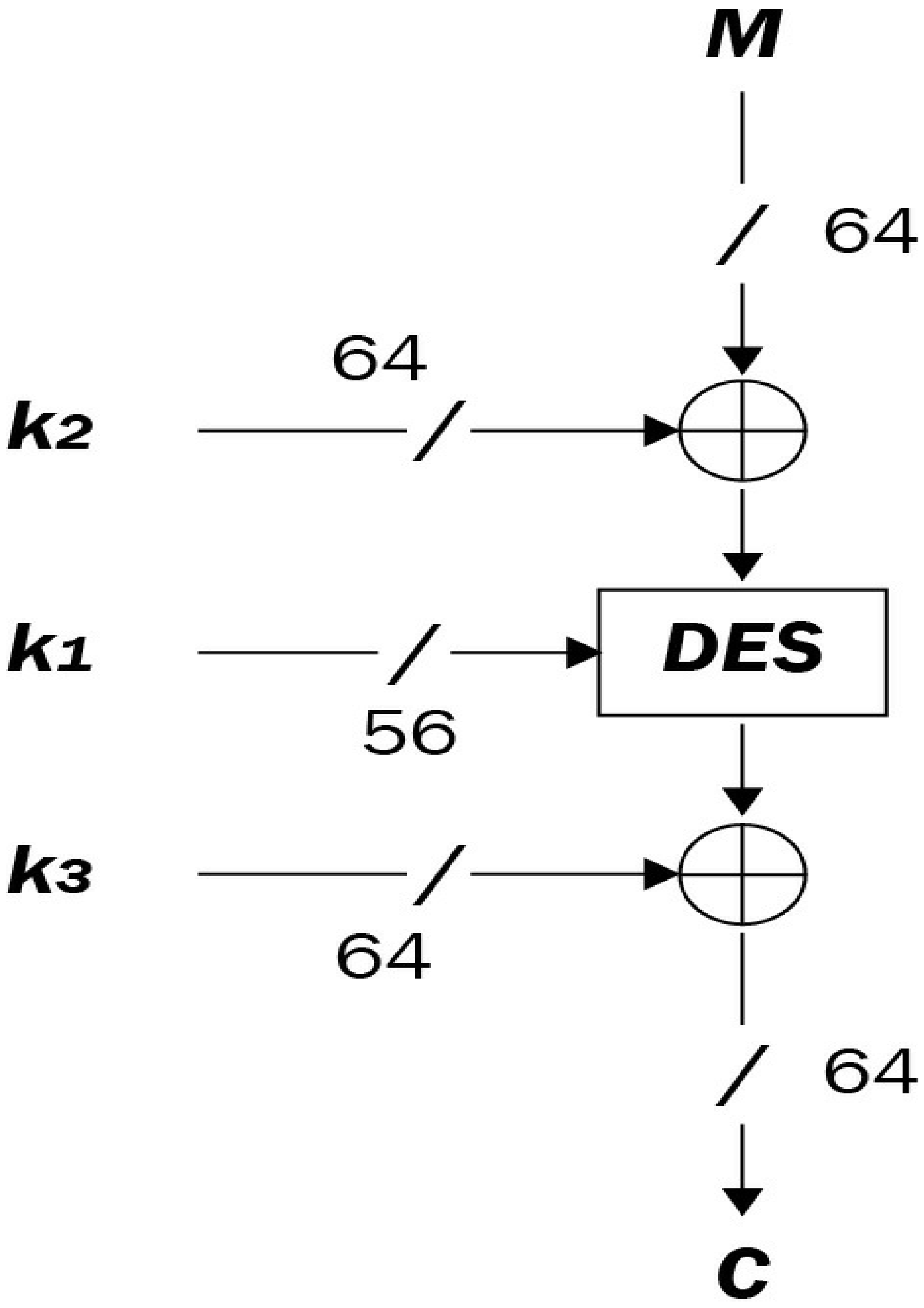

M
64
64
k2
k1
DES
56
k3
64
64
C

Figure 2.25 – DESX encryption scheme

After exploring the DES, 3DES, and DESX algorithms, we will approach another pillar of the symmetric encryption algorithm: AES. Lightweight Encryption

AES Rijndael

AES, also known as **Rijndael**, was chosen as a very robust algorithm by NIST (the US government) in 2001 after a 3-year testing period among the cryptologist community.Among the 15 candidates who competed for the best algorithm, there were five finalists chosen: **MARS** (**IBM**), **RC6** (**RSA Laboratories**), **Rijndael** (**Joan Daemen and Vincent Rijmen**), **Serpent** (**Ross Anderson and others**), and **Twofish** (**Bruce Schneier and others**). All the candidates were very strong but, in the end, Rijndael was a clear winner.The first curious question is about its name: how is Rijndael pronounced?It's dubiously difficult to pronounce this name. From the web page of the two authors, we can read that there are a few ways to pronounce this name depending on the nationality and the mother tongue of who pronounces it.Just to start, I can say that AES is a block cipher, so it can be performed in different modes: ECB (already seen in DES), **Cipher Block Chaining** (**CBC**), **Cypher Cipher Feedback Block** (**CFB**), **Output Feedback Block** (**OFB**), and **Counter** (**CTR**) mode. We will see some better differences between implementations in this section.AES can be performed using different key sizes: 128-, 192-, and 256-bit. NIST's competition aimed to find an algorithm with some very strong characteristics, such as it should be operating in blocks of 128 bits of input or it should be able to be used on different kinds of hardware, from 8-bit processors (used also in a smart card) to 32-bit architectures, commonly adopted in personal computers. Finally, it should be fast and very robust.Under certain conditions (that you will discover later), I think this is one of the best algorithms ever; indeed, I have chosen to implement AES 256 in our **Crypto Search Engine** (**CSE**). We will see CSE again in *Chapter 9*, *Crypto Search Engine*. At Cryptolab we currently adopt AES to secure the symmetric encryption of data encrypted and transmitted between virtual machines that encrypt and store data.

Description of AES

Discussing AES would alone require a dedicated chapter. In this section, I provide an overview of the algorithm. For those of you interested in knowing more, you can refer to the documentation presented by NIST (published on November 26, 2001) reported in the document titled *FIPS PUB 197*. I am limited in this chapter to describing the algorithm at just a high level and will give my comments and suggestions.Most importantly, to avoid any confusion, I will analyze AES in a different manner not found in other papers. My analysis of AES will be based on the subdivision of the algorithm into different steps. I have called these steps **Key Expansion** and **First Add Round Key**; then, as you will see later on, each step is divided into four sub-steps, called **SubBytes transformation**, **ShiftRows transformation**, **MixColumn**, and **AddRoundKey**. The important thing is to understand the scheme of the algorithm, then each round works similarly, and you can easily be guided to understand the mechanism of 10 rounds for a 128-bit key, 12 rounds for 192 bits, and 14 rounds for 256 bits. **Key Expansion** (**KE**) works as follows:

- The fixed key input of 128 bits is expanded into a key length depending on the size of AES: 128, 192, or 256.
- Then, the **[K_1]**, **[K_2]**,…**[K_r]** sub-keys are created to encrypt each round (generally adding **XOR** to the round).
- AES uses a particular method called Rijndael's *key schedule* to expand a short master key to a certain number of round keys.

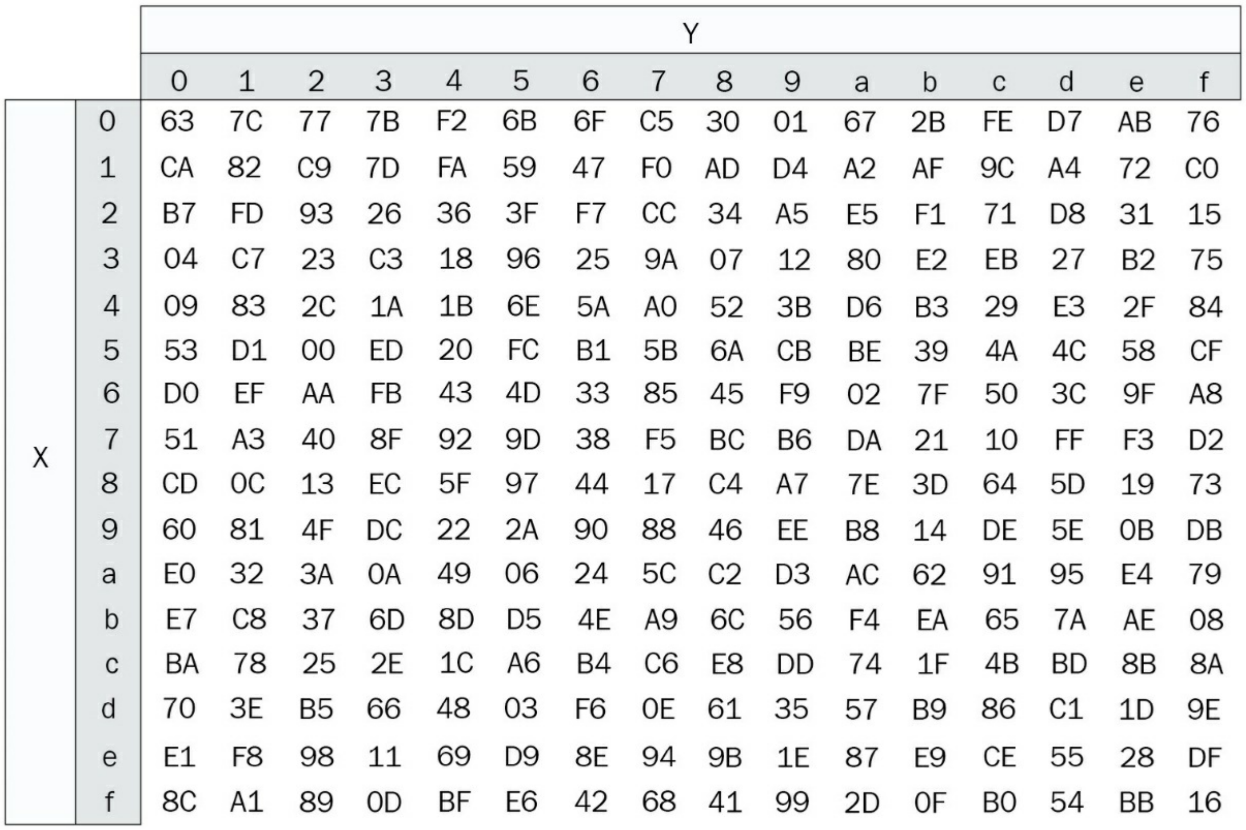
First Add Round Key (**F-ARK**) works as follows.It is the first operation. The algorithm takes the first key, **[K_1]**, and adds it to **AddRoundKey:** using a bitwise **XOR** of the current block with a portion of the expanded key.**Rounds R_1 to R_{n-1}** work as follows.Each round (except the last one) is divided into four steps called layers consisting of the following:

- **SubBytes** (**SB**) **transformation**: This step is a fundamental non-linear step, executed through a particular S-Box (we have already seen how an S-Box works in DES). You can see the AES S-Box in the following figure.
- **ShiftRows** (**SR**) **transformation**: This is a scrambling of a bit that

causes diffusion on multiple rounds.

- **MixColumns** (**MC**) **transformation**: This step has a similar scope to SR but is applied to the columns.
- **AddRoundKey** (**ARK**): The round key is **XOR**ed with the result of the previous layer.

The following figure represents S-Box Rijndael expressed in hexadecimal notation:



		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
X	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	25	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 2.26 – S-Box Rijndael

The last round runs all the operations of the previous rounds except for layer 3: **MC**. After all the earlier-mentioned processes are run for each round **n** times (depending on the size of the key: 14 rounds if the key is 256 bits), AES encryption obtains the ciphertext, **(C)**, as shown in the following figure:

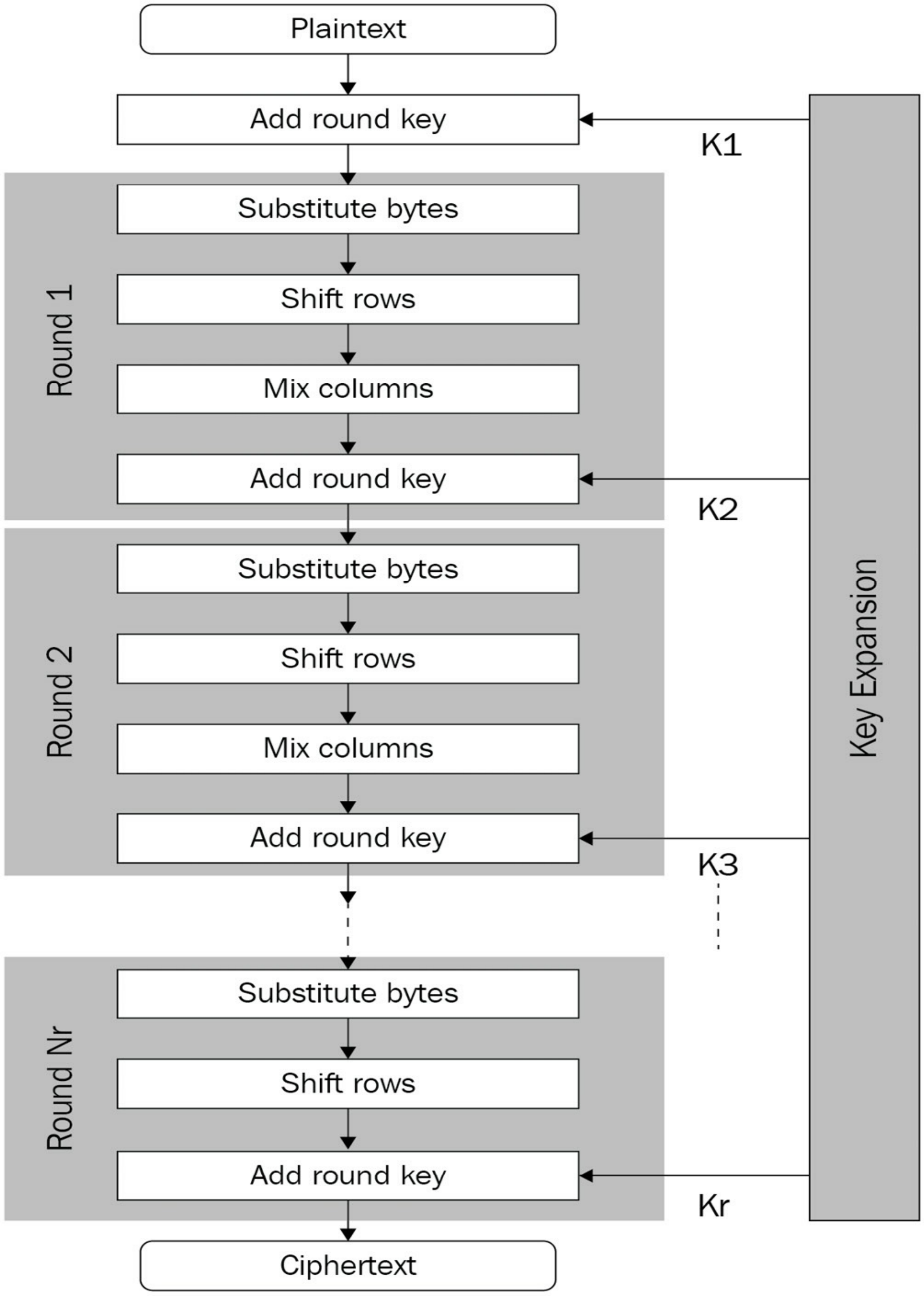

Plaintext
Add round key
K1
Round 1
Substitute bytes
Shift rows
Mix columns
Add round key
K2
Round 2
Substitute bytes
Shift rows
Mix columns
Add round key
K3
Key Expansion
Round Nr
Substitute bytes
Shift rows
Add round key
Kr
Ciphertext

Figure 2.27 – Encryption scheme in AES

Thus, we can re-schematize the entire process of AES encryption in a mathematical function as follows:

$$\begin{tikzcd}
(P) \arrow[d] \\
(KE) \arrow[d] \\
(F\text{-}ARK) \arrow[d] \\
\begin{array}{c} Kr \oplus[(R1\sim (Rr-1))] \\ {[(SB) \sim(SR)\sim(MC)\sim(ARK)]} \end{array} \arrow[d] \\
\begin{array}{c} Ki \oplus[(Rr] \\ {[(SB) \sim(SR)\sim(ARK)]} \end{array} \arrow[d] \\
(C)
\end{tikzcd}$$

Figure 2.28 – Re-schematizing an AES flow chart with mathematical functions

As you can notice from the proposed scheme in the preceding figure, we have the following:

- **K_r [(R_1~ (R_{r-1}))]** represents all the mathematical processes performed between each round key, **(K_r)**, **XOR**ed with *the inside functions* of each rounds, starting from the first round (after the F-ARK to the last round (excluded)). So, inside **[(R1~ (Rr-1))]**, we find the **[(SB) ~(SR)~(MC)~ (ARK)]** functions.
- In the last round, as you can notice, MC is not present.

In the following figure, you can see the entire process of encryption and decryption in AES:

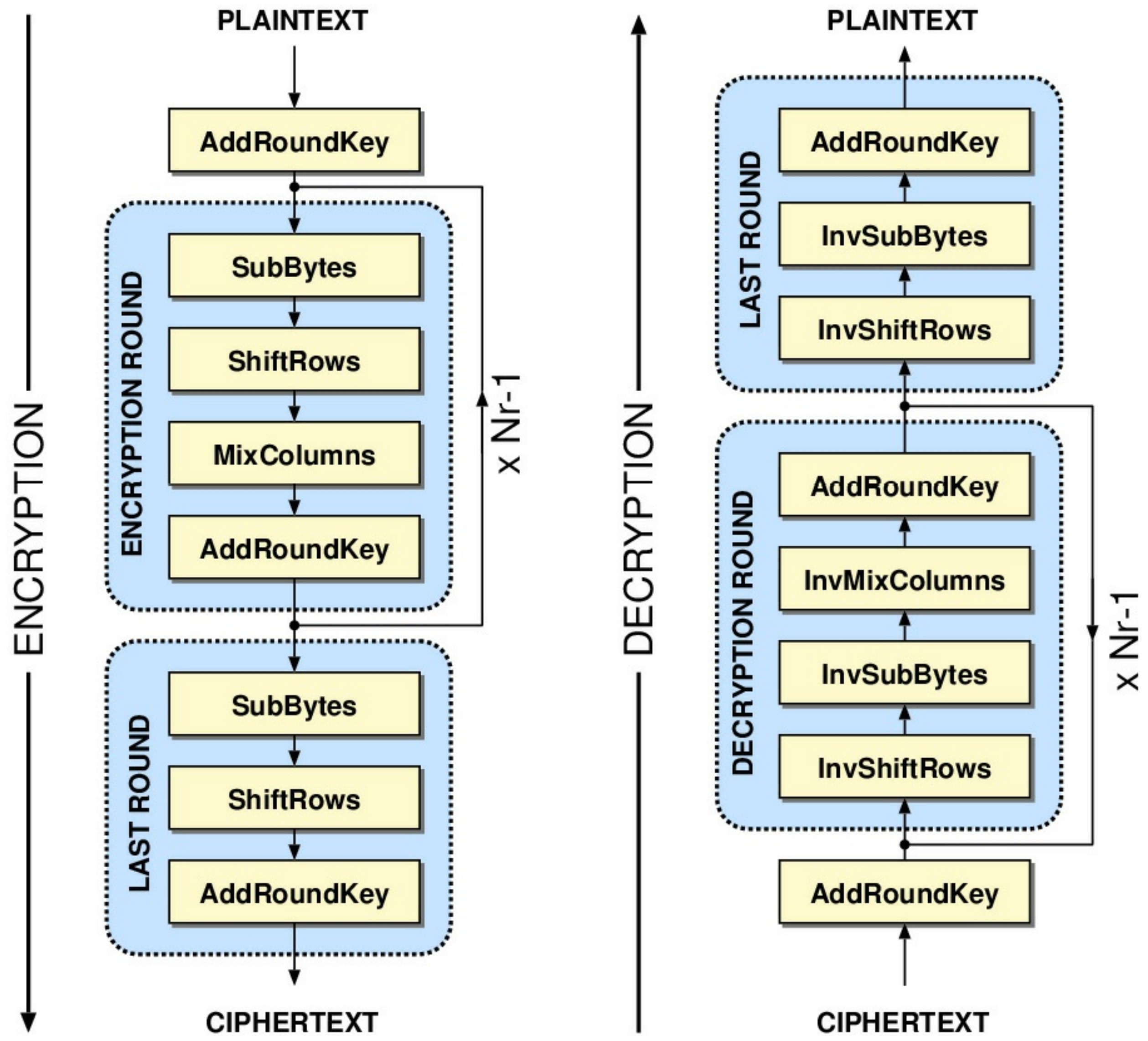


Figure 2.29 – Encryption and decryption scheme in AES

After schematizing the AES operations of encryption and decryption, we now analyze the attacks and vulnerabilities on this algorithm.

Attacks and vulnerabilities in AES

The NSA and NIST publications deemed AES as invulnerable to any kind of known attack.However, AES has its vulnerabilities; in fact, every system that can be implemented has vulnerabilities.

Important Note

Recall the NIST document reporting the possible vulnerabilities of AES (documented on October 2, 2000). You can read it at https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security.

In the NIST document, it states *Each of the candidate algorithms was required to support key sizes of 128, 192 and 256 bits. For a 128-bit key size, there are approximately 340,000,000,000,000,000,000,000,000,000,000,000,000 (340 followed by 36 zeros) possible keys*.However, even though theoretically AES remains unbreakable, **(-x%)**, using all the brute force in the world (you will see a computational analysis of AES in *Chapter 9*, *Crypto Search Engine*), it is still always possible to find a breach in any algorithm. It is common to find breaches in the implementation stage. Indeed, pay attention to what happens if you implement, for instance, AES with ECB mode. We have already seen ECB mode in DES. This basic implementation consists of dividing the plaintext into blocks and for each block of plaintext, **P**, calculating the ciphertext, **C**: `C = Encr (P)` You can see the scheme of ECB mode encryption in the following figure:

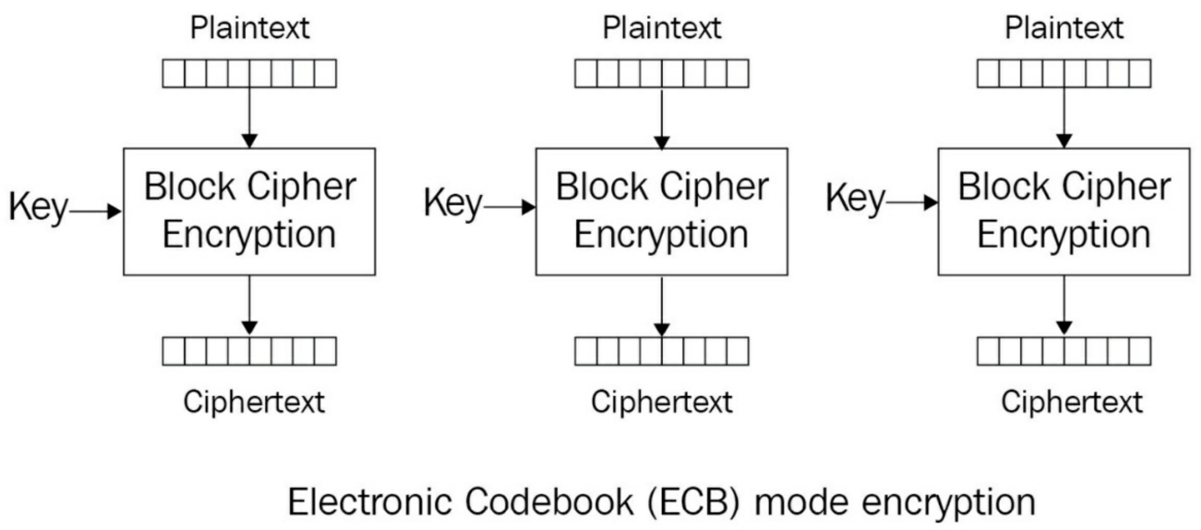


Figure 2.30 – ECB mode encryption

If AES were implemented in EBC mode, as you can see in the preceding figure, in the middle (between the original and the one encrypted with another mode), there could be serious issues. For instance, in the following *ECB of Cervino mountain*, it's possible to recognize the content even though it's encrypted:

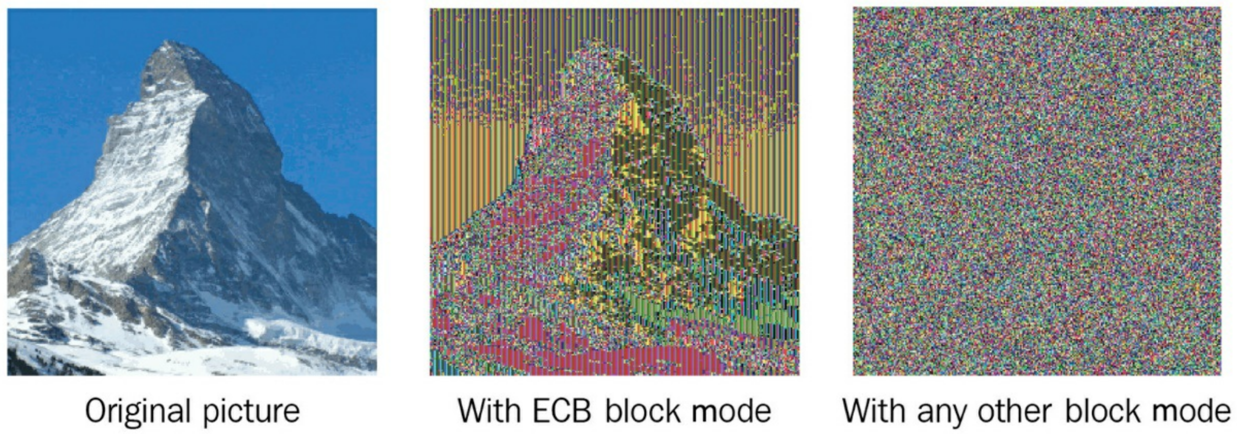


Figure 2.31 – The original picture of Cervino mountain, encrypted with ECB (failure) mode, and encrypted with another block mode

In other words, ECB block mode vanishes the encryption effect, which should have been the same as the third image (encrypted with another **block mode**).With another attack on ECB, known as **block-reply**, knowing a plaintext-ciphertext pair, even without knowing the key, it's possible for someone to repeatedly resend the known ciphertext.Now, an interesting example of this implication given by ECB mode is presented by *Christopher Swenson* in his book *Modern Cryptanalysis*. If Eve (the attacker) tries to trick Bob and Alice during the phase of information exchange, Eve can re-send the block of known plaintext with considerable advantage to herself.For example, consider this hypothetical scenario related to the ECB attack mentioned previously.Alice owns a bank account, and she goes to an ATM to withdraw money. It is assumed that the communication between Alice and the bank via the ATM is encrypted, and we suppose it would be encrypted using AES/ECB mode.So, the encrypted message between Alice (with the key **[K]**) and the bank is as follows:

- ATM: Encrypt **[K]** (name: Alice Smith, account number: 123456, amount: $200).
- Let's say that this message encrypted with AES comes out in this form:

```
CF A3 1C F4 67 T3 2D M9……
```

- Answer from the bank after having checked the account: **[OK]**.

If Eve is listening to the communication and intercepts the encrypted message, she could just repeat the operation several times until she steals all of Alice's money from the account. The bank would just think that Alice is making several more ATM withdrawals, and if no action is taken against this attack, the victim (Alice) will have just lost all the money in her account. This trick works because Eve resends the same message copied several times. If the **[K]** key doesn't change for every session of encryption, Eve can attempt this attack. A very efficient solution is to accept only different cryptograms for each session, which gets rid of the use of symmetric encryption for multiple transmissions. Otherwise, in order to prevent this kind of attack, one of the implementations of AES (just like other block ciphers) is CBC Cipher Block Chaining.CBC performs the block encryption generating an output based on the values of the previous blocks.We will see this implementation again later on, in *Chapter 9*, *Crypto Search Engine*, in the *Computational Analysis on CSE* section, when I present CSE in which we have implemented AES encryption in CBC mode.Here I'll just explain how it works.The CBC method uses a 64-bit block size for plaintext, ciphertext, and the initialization vector, **IV**. Essentially, **IV** is a random number, sometimes called *salt*, **XOR**ed to the plaintext in order to compute the block.Just remember the following:

- **E** = Encryption
- **D** = Decryption
- **C** = Ciphertext

Encryption works as follows:

- Calculate the initial block, **C**$_0$, taking the first block of the plaintext, **(P**$_0$**)**, **XOR**ing with the initialization vector (**IV**):

```
C0 = E (P0 IV)
```

- Each successive block is calculated by **XOR**ing the previous ciphertext block with the plaintext block and encrypting the result:

`Ci = E ( Pi Ci-1)` **Decryption works as follows**:

- To obtain the plaintext, **(P_0)**, combine **XOR** between the decryption of the first block of ciphertext received **(C_0)** and the initialization vector (**IV**):

`P0 = D (C0) IV`

- To obtain all the other plaintext, **(P_i)**, we have to perform an **XOR** between the decryption of the ciphertext received, **(C_i)**, and all the other ciphertext excluding the first one:

`Pi= D (Ci) Ci-1` In the following figure, the scheme of CBC mode encryption is represented:

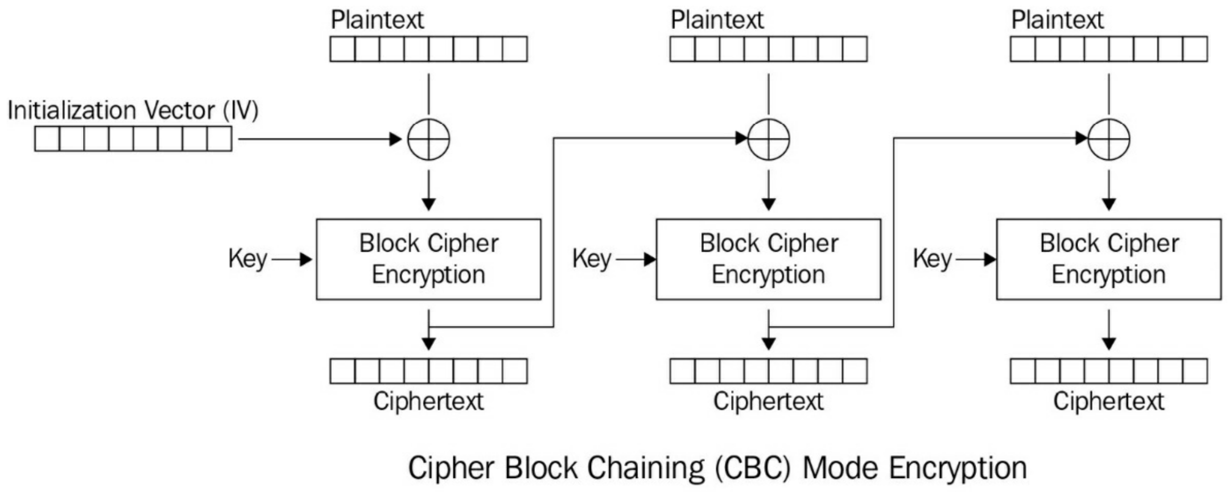


Figure 2.32 – Scheme of CBC mode encryption

AES is a robust symmetric algorithm, and until 2009, the only successful attacks were so-called side-channel attacks. These kinds of attacks are mostly related to the implementation of AES in some specific applications.Here is a list of side-channel attacks:

- **Cache attack**: Usually, some information is stored in a memory cache (a kind of memory of second order in the computer); if the attacker monitors cache access remotely, they can steal the key or the plaintext. To avoid that, it is necessary to keep the memory cache clean.

- **Timing attack**: This is a method that exploits the time to perform encryption based on the correlation between the timing and values of the parameters. If an attacker knows part of the message or part of the key, they can compare the real and modeled executed times. Essentially, it could be considered a *physical* attack on a bad implementation of the code much more than a logical attack. Anyway, this attack is not only referred to as AES but also RSA, D-H, and other algorithms too, which rely on the correlation of parameters.
- **Power monitoring attack**: Just like the timing attack, there could be potential vulnerabilities inherent to hardware implementation. You can find an interesting attack experiment at the following link relating to the correlation of the power consumption of an AES 128-bit implementation on Arduino Uno. The attack affected the **ARK** and **SB** functions of this algorithm, gaining the full 16-byte cipher key monitoring the device's power consumption. For hardware lovers, this is an exciting attack:

https://www.tandfonline.com/doi/full/10.1080/23742917.2016.1231523

- **Electromagnetic attack**: This is another kind of attack performed on the implementation of the algorithm. One attack was attempted on FPGA measuring the radiation that emanated from an antenna through an oscilloscope.

Finally, the real problem of AES is the exchanging of the key. This being a symmetric algorithm, Alice and Bob must agree on a shared key in order to perform the encryption and decryption required. Even if AES's few applications could be implemented without any key exchange, most need asymmetric algorithms to make up for the lack of key transmission. We will better understand this concept in the next chapter when we explore asymmetric encryption. Moreover, we will see an application that doesn't need any key exchange in *Chapter 9*, *Crypto Search Engine*, when analyzing the CSE.

Summary

In this chapter, you learned about symmetric encryption. We have explored the Boolean operations necessary for understanding symmetric encryption, KE, and S-Box functionality. Then, we deep-dived into how simple DES, DES, 3DES, and DESX work and their principal vulnerabilities and attacks.After these topics, we analyzed AES (Rijndael), including its implementation schema and the logic of the steps that make this algorithm so strong. Regarding the vulnerabilities and attacks on AES, you have understood how the difference between ECB mode and CBC mode can make it vulnerable to block cipher implementation attacks.Finally, we explored some of the best-known side-channel attacks valid for most cryptographic algorithms.These topics are essential because now you have learned how to implement a cryptographic symmetric algorithm, and you have more familiarity with its peculiarities. We will see many correlations with this part in the next chapters. *Chapter 9*, *Crypto Search Engine*, will explain CSE, which adopts AES as one of the algorithms for the transmission of encrypted files in the cloud. Now that you have learned about the fundamentals of symmetric encryption, it's time to analyze asymmetric encryption.

3 Asymmetric Encryption

Asymmetric encryption means using different pairs of keys to encrypt and decrypt a message. A synonym of asymmetric encryption is public/private key encryption, but there are some differences between asymmetric and public/private key encryption, which we will discover in this chapter. Starting with a little bit of history of this revolutionary method of encryption/decryption, we will look at different types of asymmetric algorithms and how they help secure our credit cards, identity, and data.In this chapter, we are going to cover the following topics:

- Public/private key and asymmetric encryption
- The Diffie-Hellman key exchange and the related man-in-the-middle problem
- RSA and an interesting application for international threats
- Introduction to conventional and unconventional attacks on RSA
- **Pretty Good Privacy** (**PGP**)
- ElGamal and its possible vulnerabilities

Let's dive in!

Introduction to asymmetric encryption

The most important function of private/public key encryption is exchanging a key between two parties, along with providing secure information transactions.To fully understand asymmetric encryption, we must understand its background. This kind of cryptography is particularly important in our day-to-day lives. This is the branch of cryptography that's deputed to cover our financial secrets, such as credit cards and online accounts; to generate the passwords that we use constantly in our lives; and, in general, to share sensitive data with others securely and protect our privacy.Let's learn a little bit about the history of this fascinating branch of cryptography.The story of asymmetric cryptography begins in the late 1970s, but it advanced in the 1980s when the advent of the internet and the digital economy started to introduce computers to family homes. The late 1970s and 1980s was the period in which Steve Jobs founded Apple Inc. and was during the Cold War between the USA and the USSR. It was also a period of economic boom for many Western countries, such as Italy, France, and Germany. And finally, it was the period of the advent of the internet. The contraposition of the two blocs, Western and Eastern, with US allies on one side and the Soviet bloc on the other side, created opposing networks of spies that had their fulcrum in the divided city of Berlin. During this period, keys being exchanged in symmetric cryptography reached the point that the US Government's **Authority for Communications Security** (**COMSEC**), which is responsible for transmitting cryptographic keys, transported tons of keys every day. This problematic situation degenerated to a breaking point. Just to give an example, with the DES algorithm in the 1970s, banks dispatched keys via a courier that were handed over in person. The **National Security Agency** (**NSA**) in America struggled a lot with the key distribution problem, despite having access to the world's greatest computing resources. The issue of key distribution seemed to be unsolvable, even for big corporations dedicated to solving the hardest problems related to the future of the world, such as RAND – another powerful institution created to manage the problems of the future and to prevent breakpoint failures. I think that, sometimes, a breakpoint is just a way to clear up the situation instead of just ignoring it. Sometimes, issues have different ways they can be solved. In the case of asymmetric encryption, no amount of government money nor supercomputers with infinite computation and multiple brains at their service could solve a problem that, at a glance, would appear rather easy to solve.Now that you have an idea of the main problem that asymmetric encryption solves, which is the key exchange (actually, we will see that in RSA, this problem gets translated into the direct transmission of the message), let's go deeper to explore the pioneers involved in the history of this extremely intriguing branch of cryptography.

The pioneers

Cryptographers can often appear to be strange people; sometimes, they could be introverts, sometimes extroverts. This is the case with *Whitfield Diffie*, an independent freethinker, not employed by the government or any of the big corporations. I met Diffie for the first time at a convention in San Francisco in 2016 while he was discussing cryptography with his famous colleagues, *Martin Hellmann* and *Ronald Rives*. One of the most impressive things that remained fixed in my mind was his elegant white attire, counterposed by his tall stature and long white hair and beard. Similar to an ever-young guy still in the 1960s, someone whose contemporary could be an agent at the Wall Street Stock Exchange or a holy man in India. He is one of the fathers of modern cryptography, and his name will be forever imprinted in the history of public/private key encryption. Diffie was born in 1944 and graduated from MIT in Boston in 1965. After his graduation, he was employed in the field of cybersecurity and later became one of the most authentic independent cryptographers of the 1970s. He has been described as *cyberpunk*, in honor of the *new wave* science fiction movement of the 1960s and 1970s, where cybernetics, artificial intelligence, and hacker culture combined into a dystopian futuristic setting that tended to focus on a *combination of low life and high tech*.Back in the 1960s, the US Department of Defense began funding a new advanced program of research in the field of communication called **Defense Advanced Research Projects Agency** (**DARPA**), also called **ARPA**. The main ARPA project was to connect military computers to create a more resilient grade of security in telecommunications. The project intended to prevent a blackout of communications in the event of a nuclear attack, but also, the network allowed dispatches and information to be sent between scientists, as well as calculations that had been performed, to exploit the spare capacity of the connected computers. **ARPANET** started officially in 1969 with the connection of only four sites and grew quickly so much that in 1982, the project spawned the internet. At the end of the 1980s, many regular users were connected to the internet, and thereafter their number exploded.While ARPANET was growing, Diffie started to think that one day, everyone would have a computer, and with it, exchange emails with each other. He also imagined a world where products could be sold via the internet and real money was abandoned in favor of credit cards. His great consideration of privacy and data security led to Diffie being obsessed with the problem of how to communicate with others without having any idea who was at the opposite end of the cable. Moreover, encrypting messages and documents is often done when sending highly valuable information; data encryption was starting to be used by the general public to hide information and share secrets with others. This was the time when the usage of cryptography became common, and it was not just for the military, governments, or academics.The main issue to solve was that if two perfect strangers meet each other via the internet, how would it be possible to encrypt/decrypt a shared document without exchanging any additional information except for the document itself, which is encrypted/decrypted through mathematical parameters? This is the key exchange problem in a nutshell.One day in 1974, Diffie went to visit IBM's *Thomas Watson* laboratory, where he was invited to give a speech. He spoke about various strategies for attacking the key distribution problem but the people in the room were very skeptical about his solution. Only one shared his vision: he was a senior cryptographer for IBM who mentioned that a Stanford professor had recently visited the laboratory and had a similar vision about the problem. This man was Martin Hellman.Diffie was so enthusiastic that the same evening, he drove to the opposite side of America to Palo Alto, California to visit Professor Hellman. The collaboration between Diffie and Hellman will remain famous in cryptography for the creation of one of the most beautiful algorithms in the field: the **Diffie-Hellman key exchange**.We'll analyze this pioneering algorithm in the next section.

The Diffie-Hellman algorithm

To understand the **Diffie-Hellman** (**D-H**) algorithm, we can rely on the so-called *thought experiments* or *mental representation* of a theory often used by Einstein.A thought experiment is a hypothetical scenario where you mentally transport yourself to a more real situation than in the purely theoretical way of facing an issue. For example, Einstein used a very popular thought experiment to explain the theory of relativity. He used the metaphor of a moving train observed by onlookers from different positions, inside and outside of the train.I will often apply these mental figurative representations in this book.Let's imagine that we have our two actors, Alice and Bob, who want to exchange a message (on paper) but the main post office in the city examines the contents of all letters. So, Alice and Bob struggled a lot with different methods to send a letter secretly while avoiding any intrusion; for example, putting a key inside a metallic cage and sending it to Bob. But because Bob wouldn't have the key to open the cage, Alice and Bob would have to meet somewhere first so that Alice could give the key to Bob. Again, we return to the problem of exchanging keys.After many, many attempts, it seemed to be impossible to arrive at a logical solution that would solve this problem, but finally, one day, Diffie, with Hellmann's support, found the solution, about which Hellmann later said, *"God rewards fools!"*Let's explore a mental representation of what Alice and Bob should do to exchange the key:

- **Step 1**: Alice puts her secret message inside a metallic box closed with an iron padlock and sends it to Bob, but holds on to the key herself. Now, remember that Alice locks the box using her key and doesn't give it to Bob.
- **Step 2**: Bob applies one more lock to the cage using his private key and resends the box to Alice. So, after Bob has received the box for the first time, he can't open it. He just adds one more lock to the box.
- **Step 3**: When Alice receives the box the second time, she is free to remove her padlock since the box remains secured with Bob's key, as shown in the following diagram, when Alice resends the box for the last time. Remember that the message is always inside the box. Right now, the box is only locked with Bob's padlock.
- **Step 4**: When Bob receives the box this time, he can open it, because the box only remains locked with his padlock. Finally, Bob can read the content of the message sent by Alice that has been preserved inside the box.

As you can see, Alice and Bob never met each other to exchange any padlock keys. Note that in this example, the box was sent twice from Alice to Bob, while in the algorithm, it is not:

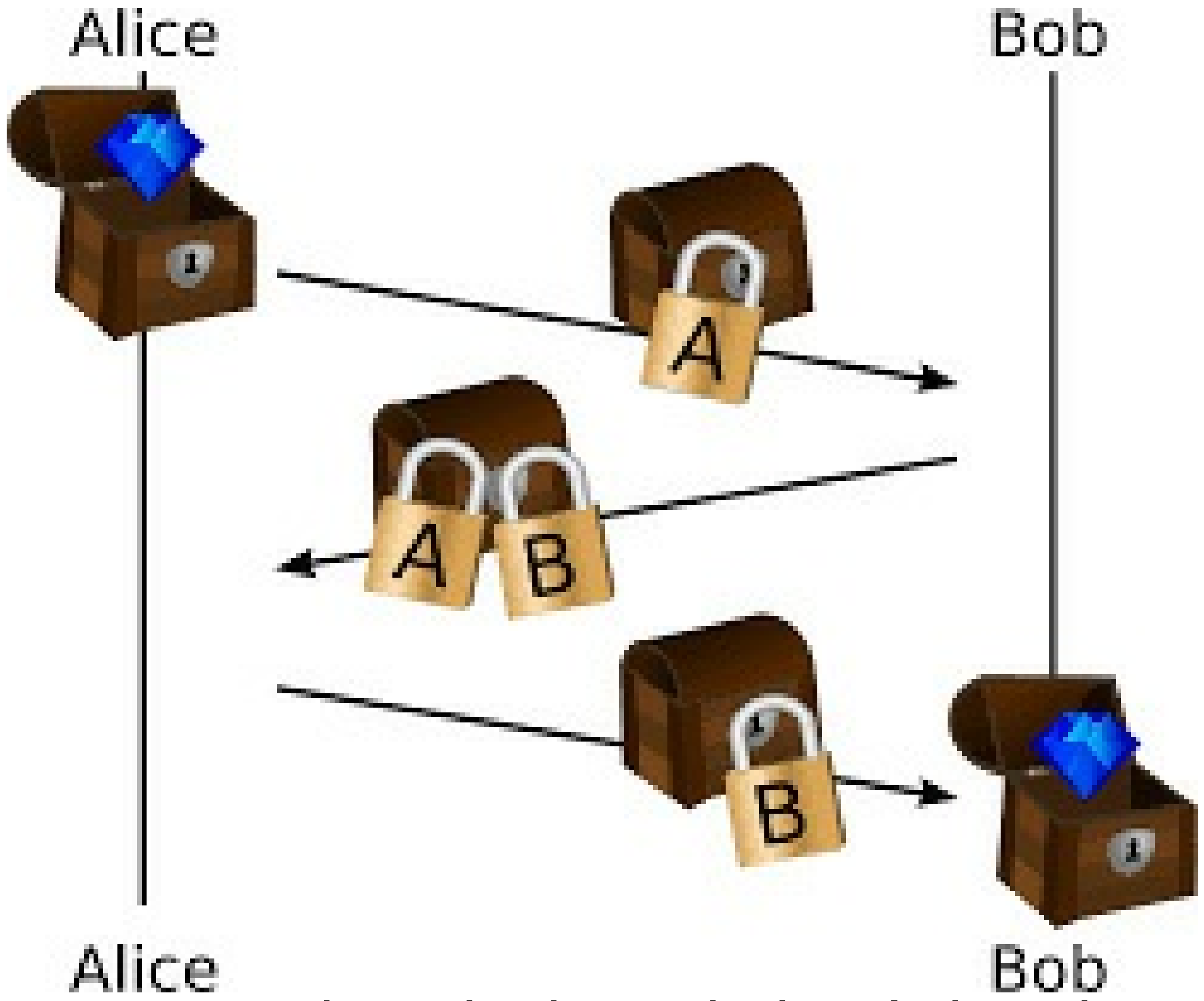


Figure 3.1 – The D-H algorithm using the Alice and Bob example

The preceding explanation and representation are not exactly what the algorithm does, but it provides a practical solution to a problem believed unsolvable: the key exchange problem.Now, we have to transpose this practical argumentation into a logical-mathematical representation. First of all, we will return to using modular mathematics while taking advantage of some particular properties of the operations made in finite fields.

The discrete logarithm

I will try to explain the math behind cryptography without using excessive notations, just because I don't want you to get confused, or to load this book with heavy mathematical dissertations. It's not within the scope of this book.When we talk about finite fields, we are considering a finite group of **(n)** integer numbers, **(Z)**, laying in a ring – let's say, **(Zn)**. This group of numbers will be subjected to all the same mathematical laws, such as operations with standard integers. Since we are working in a finite field called **(modulo n)** here, we have to consider some critical issues that involve modular mathematics. As we saw in the previous chapter, operating in *modulus* means wrapping back to the first number each time we arrive at the end of the set. This is just like the clock's math, where we wrap back to 1 when we reach 12.Essentially, remember that in a finite field, there is a *numerical period* in which the numbers and the results of the operations of the field recur. For example, if we have a set of seven integers, **{0, 1, 2, 3, 4, 5, 6}**, often abbreviated as **(Z7)**, we have all the operations that have been performed inside this finite field wrapping back inside the integers of the field.Here is a short example of operations within a finite field, **(Z7, +, x)**, of addition and multiplication. Since all the operations, **(modulo 7)**, will work, we have to consider that the numbers will wrap back to **0** each time the operation exceeds the number **7**: `1 x 1 ≡ 1 (mod 7)2 x 4 ≡ 1 (mod 7)3 + 5 ≡ 1 (mod 7)3 x 5 ≡ 1 (m` let's use this modality of counting and consider that the **=** notation is equivalent to **≡**, which is the mathematics we learned at elementary school, where **2 + 2 = 4** doesn't properly work if we consider, for example, a finite field of **modulo 3**: `2 + 2 ≡ 1 (mod 3)` From high school mathematics, we recall that the **[log a(z)]** logarithm is a function where **(a)** is the base. We have to determine the exponent to give to **(a)** to obtain the number **(z)**. So, for example, if **a = 10** and **z = 100**, we find that the logarithm is **2** and we say that **logarithm base 10 of 100 is 2**. If we use Mathematica to calculate a logarithm, we have to compose a different notation, that is, **Log[10, 100]= 2**.While working in the discrete field, things became more complicated, so instead of using a normal logarithm, we started working with a discrete logarithm.So, let's say we have to solve an equation like this: `a^[x] ≡ b (mod p)` This would be a very hard problem, even if we know the value of **(a)** and **(b)**, because there is no efficient algorithm known to solve the discrete logarithm, that is, **[x]**.

Important Note

I have used square brackets to say that **[x]** is secret. Technically **[x]** is a discrete power, but the problems of searching for discrete logarithm and discrete power have the same computational difficulty.

Let's go a little bit deeper now to explain the dynamics of this operation. Let's consider computing the following: `2^4 ≡ (x) (mod 13)` First, we calculate **2^4 = 16**, then divide **16:13 = 1** by the remainder of **3** so that **x = 3**. The discrete logarithm is just the inverse operation: `2^[y] ≡ (x) (mod 13)` In this case, we have to calculate **[y]** while knowing the base is **2**. It's possible to demonstrate that there are infinite solutions for **[y]** that generate **(x)**.Taking the preceding example, we have the following: `2^[y] ≡ 3 (mod 13) for [y]` One solution is **y = 4**, but it is not the only one.The result of **3** is also satisfied for all the integers, **(n)**, of this equation: `[y] = [y + (p-1)*n]` Let's prove **n = 1**: `2^[4+(13-1)*1] ≡ 2^16 (mod 13)   2^16 ≡ 3 (mod 13)` But it is also valid for **n = 2**: `2^[4 +(13-1)*2] ≡ 2^28 (mod 13)      2^28 ≡ 3 (mod 13)` And so on…Hence, the equation has infinite solutions for all the integers; that is, **(n ≥ 0)**: `[y] ≡ 2^[4 + 12 n] (mod 13)` There is no method yet for solving the discrete logarithm in polynomial time. So, in mathematics, as in cryptography, this problem is considered very hard to solve, even if the attacker has a lot of computation power.Finally, we have to introduce the definition and the notation of a generator, **(g)**, which is a particular number where we say that **(g)** generates the entire group, **(Zp)**. If **(p)** is a prime number, this means that **(g)** can take on any value between **1** and **p-1**.

Explaining the D-H algorithm

D-H is not exactly an asymmetric encryption algorithm, but it can be defined properly as a public/private key algorithm. The difference between asymmetric and public/private keys is not only formal but substantial. You will understand the difference better later, in the *RSA* section, which covers a pure asymmetric algorithm. Instead, D-H gets a shared key, which works to symmetrically encrypt the message, **[M]**.The encryption that's performed with a symmetric algorithm is combined with a D-H shared key transmission to generate the cryptogram, **C**: `Symmetric-Algorithm E((D-H[k]), M) = C` In other words, we use the

D-H algorithm to generate the shared secret key, **[k]**, then with AES or another symmetric algorithm, we encrypt the message, **[M]**.D-H doesn't directly encrypt the secret message; it can only determine a shared key between two parties. This is a critical point, as we will see in the next paragraph.However, for working in discrete fields and applying a discrete logarithm problem to shield the key from attackers when sharing it, Diffie and Hellman implemented one of the most robust and famous algorithms in cryptography.Let's see how D-H works:

- **Step 1**: Alice and Bob first agree on the parameters: **(g)** as a generator in the ring, **(Zp)**, and a prime number, **p (mod p)**.
- **Step 2**: Alice chooses a secret number, **[a]**, and Bob chooses a secret number, **[b]**.

Alice calculates **A ≡ g^a (mod p)** and Bob calculates **B ≡ g^b (mod p)**.

- **Step 3**: Alice sends **(A)** to Bob and Bob sends **(B)** to Alice.
- **Step 4**: Alice computes **ka ≡ B^a (mod p)** and Bob computes **kb ≡ A^b (mod p)**.

[ka = kb] will be the secret that's shared key between Alice and Bob.The following is a numerical example of this algorithm:

- **Step 1**: Alice and Bob agree on the parameters they will use; that is, **g = 7** and **(mod 11)**.
- **Step 2**: Alice chooses a secret number, **(3)**, and Bob chooses a secret number, **(6)**.

Alice calculates **7^3 (mod 11) ≡ 2** and Bob calculates **7^6 (mod 11) ≡ 4**.

- **Step 3**: Alice sends **(2)** to Bob and Bob sends **(4)** to Alice.
- **Step 4**: Alice computes **4^3 (mod 11) ≡ 9** and Bob computes **2^6 (mod 11) ≡ 9**.

The number **[9]** is the shared secret key, **[k]**, of Alice and Bob.

Analyzing the algorithm

In *Step 2* of the algorithm, Alice calculates **A ≡ g^a (mod p)** and Bob calculates **B ≡ g^b (mod p)**. Alice and Bob have exchanged **(A)** and **(B)**, the public parameters of the one-way function. A *one-way function* has this name because it is impossible, **(-x%)**, to return from the public parameter, **(A)**, to calculate the secret private key, **[a]** (and the same for **(B)** with **[b]**), for the robustness of the discrete logarithm (see the *The discrete logarithm* section).Another property of the modular powers is that we can write **B^a** and **A^b (mod p)**, as follows: `B ^a ≡ (g^b)^a (mod p)A ^b ≡ (g^a)^b (mod p)` So, for the property of modular exponentiations, we have the following: `g^(b*a) ≡ g^(a*b) (mod p)` For example, we have the following:

- Alice: **(7^6)^3 ≡ 7^(6*3) ≡ 9 (mod 11)**
- Bob: **(7^3)^6 ≡ 7^(3*6) ≡ 9 (mod 11)**

This is the mathematical trick that makes it possible for the D-H algorithm to work.Now that we have understood the algorithm, let's highlight its defects and the possible attacks that can be performed on it.

Possible attacks and cryptanalysis on the D-H algorithm

The most common attack that's performed on the D-H algorithm is the **man-in-the-middle** (**MiM**) attack.A MiM attack is when the attacker infiltrates a channel of communication and spies on it, blocks, or alters the communication between the sender and the receiver. Usually, the attack is accomplished by Eve (the attacker) pretending to be one of the two true actors in the conversation:

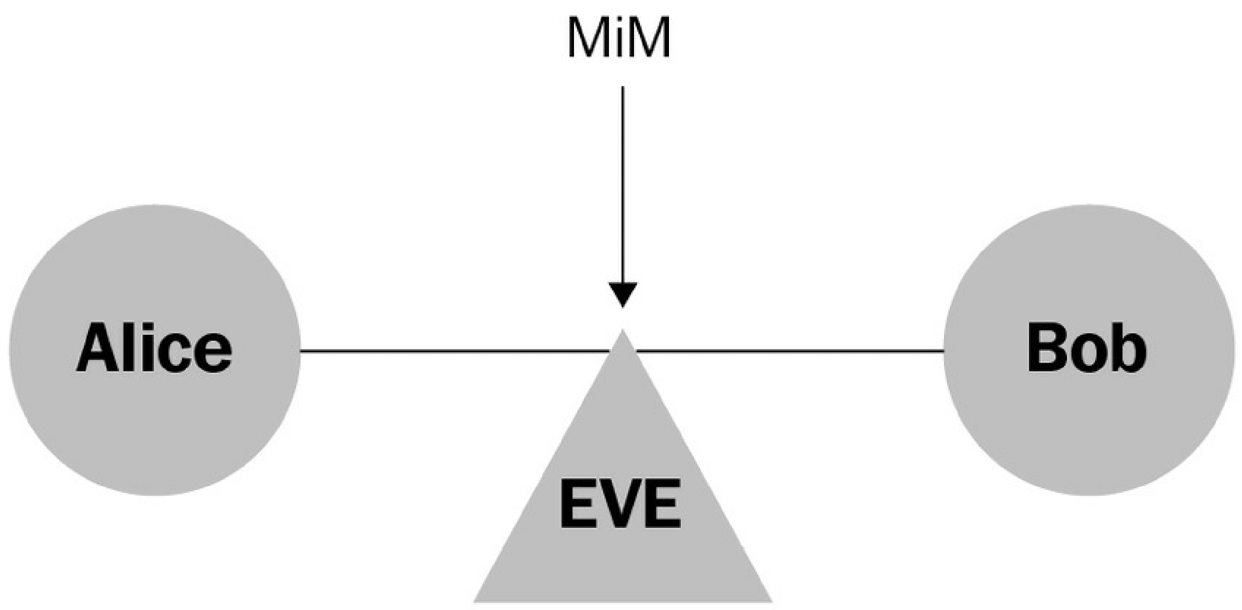


Figure 3.2 – Eve is the "man in the middle"

Recalling what happened in *Step 3* of the D-H algorithm, Alice and Bob exchanged their public parameters, **(A)** and **(B)**.Here, Alice sends **(A)** to Bob, then Bob sends **(B)** to Alice.Now, Eve (the attacker) interferes within the communication by pretending to be Alice.So, a MiM attack looks like this:

- **Step 3**: Alice sends **(A)** to Bob and Bob sends **(B)** to Alice.
- **Here, we have the MiM attack**: Eve sends **(E)** to Bob and then Bob sends **(B)** to Eve, assuming it is Alice.

Let's analyze Alice's function, **A ≡ g^a (mod p) and Bob's function B ≡ g^b (mod p)**. This passage would be crucial if it was done in normal arithmetic and not in finite fields. There is another possible attack, also known as the **birthday attack**, which is one of the most famous attacks performed on discrete logarithms. It is consolidated that among a group of people, at least two of them will share a birthday so that in a cyclic group, it will be possible to find some equal values (collisions) to solve a discrete logarithm.

Important Note

(E) here represents Eve's public parameter, which has been generated by her private key, not encryption.

Proceeding with the final part of the algorithm, you can see the effects of the MiM trick.Suppose that Alice and Bob are using D-H to generate a shared private key to encrypt the following message:Alice's message: *Bob, please transfer $10,000.00 to my account number 1234567*.After *Step 3*, in which Bob and Eve (pretending to be Alice) have exchanged their public parameters, Eve sends the modified message to Bob (intercepted from Alice), which has been encrypted with the shared key.Suppose, the encrypted message from Eve is **bu3fb3440r3nrunfjr3umi4gj57*je**.Bob receives the preceding encrypted message (supposedly from Alice) and decrypts it with the D-H shared key, obtaining some plaintext.Eve's modified message after the MiM attack is *Bob, please transfer $10,000.00 to my account number 3217654*.As you will have noticed, the account number is Eve's account. This attack is potentially disruptive.Analyzing the attack, *Step 3* is not critical (as we have said) because **(A)** and **(B)** have been communicated in clear mode, but the question is: how can Bob be sure that **(A)** is coming from Alice?The answer is, by using the D-H algorithm, Bob can't be sure that **(A)** comes from Alice and not from Eve (the attacker); similarly, Alice can't know that **(B)** comes from Bob either. In the absence of additional information about the identity of the two parties, relying only on the parameters received, the D-H algorithm suffers from this possible substitution-of-identity attack called MiMThis example shows the need for the sender (Alice) and the receiver (Bob) to have a way to be sure that they are who they say they are, and that their public keys, **(A)** and **(B)**, do come from Alice and Bob, respectively. To prevent the problem of a MiM attack and identify the users of the communication channel, one of the most widely used techniques is a digital signature. We will look at digital signatures in *Chapter 4, Introducing Hash Functions and Digital Signatures*, which is entirely dedicated to explaining these cryptographic methods.Moreover, it's possible for a public/private algorithm to identify the parties and overcome the MiM attack. In *Part 4* of this book, I will show you some public/private key algorithms of the new generation that, although not asymmetric, have multiple ways to be signed.Finally, a version of D-H can be implemented using elliptic curves. We will analyze this algorithm in *Chapter 7, Elliptic Curves*.

RSA

Among the cryptography algorithms, RSA shines like a star. Its beauty is equal to its logical simplicity and hidden inside is such a force that after 40 years, it's still used to protect more than 80 percent of commercial transactions in the world.Its acronym is made up of the names of its three inventors – **Rivest, Shamir, and Aldemar. RSA** is what we call the perfect asymmetric algorithm. Actually, in 1997, the CESG, an English cryptography agency, attributed the invention of public-key encryption to James Allis in 1970 and the same agency declared that in 1973, a document was written by Clifford Cocks that demonstrates a similar version of the RSA algorithm.The essential concept of the asymmetric algorithm is that the keys for encryption and decryption are different.Recalling the analogy to padlocks I made in the *The Diffie-Hellman algorithm* section, when I described the D-H algorithm, we saw that anybody (not just Alice and Bob) could lock the box with a padlock. This is the true problem of MiM because the padlock can't be recognized as specifically belonging to Bob or Alice.To overcome this problem, another interesting mental experiment can be done using a similar analogy but making things a little different.Suppose that Alice makes many copies of her padlock, and she sends these copies to every postal office in the country, keeping the key that opens all the padlocks in a secret place.If Bob wishes to send a secret message to Alice, he can go to a postal office and ask for Alice's padlock, put the message inside the box, and lock it.In this way, Bob (the sender), from the moment he locks the box, will be unable to unlock it, but when Alice receives the box, she will be able to open it with her unique secure key.Basically, in RSA (as opposed to D-H), Bob encrypts the message with Alice's public key. After the encryption process, even Bob is unable to decrypt the message, while Alice can decrypt it using her private key.This was the step that transformed the concept of asymmetric encryption from mere theory to practical use. RSA discovered how to encrypt a message with the public key of the receiver and decrypt it with the private key. To make that possible, RSA needs a particular mathematic function that I will explain further later on when we explore the algorithm in detail.As we mentioned previously, there were three inventors of this algorithm. They were all researchers at MIT, Boston, at the time. We are talking about the late 1970s. After the invention of the D-H algorithm, Ronald Rivest was extremely fascinated by this new kind of cryptography. He first involved a mathematician, Leonard Adleman, and then another colleague from the Computer Science department, Aid Shamir, who joined the group. What Rivest was trying to achieve was a mathematical way to send a secret message encrypted with a public key and decrypt it with the private key of the receiver. However, in D-H, the message can only be encrypted once the key has been exchanged, using the same shared key. Here, the problem was to find a way to send the message that had been encrypted with a public key and decrypted through the private key. But as I've said, it needed a very particular inverse mathematical function. This is the real added value of the RSA invention that we are going to discover shortly.The tale of this discovery, as Rivest told it, is funny. It was April 1977 when Rivest and Adleman met at the home of a student for Easter. They drank a little too much wine and at about midnight, Rivest went back home. He started to think over the problem that had been tormenting him for almost a year. Laying on his bed, he opened a mathematics book and discovered the function that could be perfect for the goal that the group had.The function he found was a particular inverse function in modular mathematics related to the factorization problem.As I introduced in *Chapter 1*, *Deep Diving into Cryptography*, the problem of factorizing a large number made by multiplying two big prime numbers is considered very hard to solve, even for a computer with immense computation power.

Explaining RSA

To understand this algorithm, we will consider Alice and Bob exchanging a secret message.Let's say that Bob wants to send a secret message to Alice, given the following:

- **M**: The secret message
- **e**: A public parameter (usually a fixed number)
- **c**: The ciphertext or cryptogram
- **p**, **q**: Two large random prime numbers (the private keys of Alice)

The following is the public and private key generation. As you will see, the core of RSA (its magic function) is generating Alice's private key, **[d]**.**Key generation**:Alice's public key, **(N)**, is given by the following code: `N = p*q` As we mentioned earlier, multiplying two big prime numbers makes **(N)** very difficult to factorize, and makes it generally very difficult for an attacker to find **[p]** and **[q]**. Alice's private key, **[d]**, is given by the following code: `[d] * e ≡ 1 (mod[p-1]*[q-1])` Bob performs the encryption: `c ≡ M^e (mod N)` Bob sends the ciphertext, **(c)**, to Alice. She can now decrypt **(c)** using her private key, **[d]**: `c^d ≡ M (mod N)` And that's it!

Note

The bold elements – **M**, **d**, **p**, and **q** – in the algorithm are protected and secret.

Numerical example: `M = 88e = 9007p = 101q = 67N = 6767` **Step 1**: Bob's encryption is as follows: `88^9007 ≡ 6621 (mod 6767)` Alice receives a cryptogram, that is, **c = 6621**.**Step 2**: Alice's decryption is as follows: `9007* d ≡ 1 (mod (101-1)*(67-1))d = 39436621^3943 ≡ 88 (mo` you can see, the secret message, **[M] = 88**, comes back from Alice's private key, **[d] = 3943**.

Analyzing RSA

There are several elements to explain but the most important is to understand why this function, which is used for decrypting **(c)** and obtaining **[M]**, works: `M ≡ c^[d] (mod N)` This is *Step 2*; that is, the decryption function. I have just inverted the notation by putting **[M]** on the left. The reason it works is hidden in the key generation equation: `[d] * e ≡ 1 (mod [p-1]*[q-1])` **[d]** is Alice's private key. For Euler's theorem, the function will probably be verified because the numbers **[p]** and **[q]** are very big and **[M]** is probably a co-prime of **(N)**. If this equation is verified, then we can rewrite the encryption stage as follows: `(M^e)^d (mod N)` For the properties of the powers and Euler's theorem, we have the following: `M^(e*d) (mod N)de ≡ 1 (mod (p-1)*(q-1))` That is the same as writing **M^1 = M (mod N)**. So, by inserting **[d]** inside the decryption stage, Alice can obtain **[M]**.

Conventional attacks on the algorithm

All the attacks that will be explained in the first part of this section are recognized and well known. That is why we are talking about conventional attacks on RSA.The first three methods of attack on RSA are related to the **(mod N)** public parameter. To perform an attack on **N = p*q**, the attacker could do the following:

- Use an *efficient* algorithm of factorization to discover **p** and **q**.
- Use new algorithms that, under certain conditions, can find the numbers.
- Use a quantum computer to factorize **N** (in the future).

Let's analyze the following three cases:

- In the first case, an efficient algorithm of factorization is not yet known. The most common methods are as follows:
- The general number field sieve algorithm
- The quadratic sieve algorithm
- The Pollard algorithm
- In the second case, if (**n**) is the number of digits of **N = p*q** and the attacker knows the first (**n/4**) digits or the last (**n/4**) digits of **[p]** or **[q]**, then it will be possible to factorize **(N)** in an efficient way. Anyway, there is a very remote possibility of knowing it. For example, if **[p]** and **[q]** have 100 digits and the first (or the last) 50 digits of **[p]** are known, then it's possible to factorize **N**.

For more information, you can refer to Coppersmith's method of factorization. More cases related to Coppersmith attacks, as explained later in this section, are where the exponents, **(e)** or **[d]**, and even the plaintext, **[M]**, are too short.

- If an attacker uses a quantum computer, it will be theoretically possible to factorize **N** in a short time with Shor's algorithm, and I am convinced that in the future, other, more efficient quantum algorithms will arise. I will explain this theory in more detail in *Chapter 8, Quantum Cryptography*, where we talk about quantum computing and Q-cryptography.

Finally, if we have a very short piece of plaintext, **[M]**, and even the exponent, **(e)**, is short, then RSA could be breakable. This is because the power operation, **M^e**, remains inside modulo **N**. So, in this phase of encryption, let's say we have the following: `M^e < N` Here, it's enough to use the **e-th** root of **(c)** to find **]M[**.

Important Note

I have used open brackets, **]M[**, to denote that the message has been decrypted.

Numerical example: `M = 2e = 3N = 772^3 ≡ 8 (mod 77)` Since **e = 3**, by performing a simple cubic root, √, we can obtain the message in cleartext: `8^(1/3) = 2` Here, we are working in linear mathematics and no longer in modular mathematics.A way to overcome this problem is to lengthen the message by adding random bits to it. This method is very common in cryptography and cybersecurity and is known as **padding**.There are different ways to perform padding, but here, we are talking about **bit padding**. As we covered in *Chapter 1, Deep Diving into Cryptography,* we can use ASCII code to convert text into a binary system, so the message, **[M]**, is a string of bits. If we add random bits (usually at the end, though they could also be added at the start), we will obtain something like this: `... | 1011 1001 1101 0100 0010 0111 0000 0000 |` As you can see, the bold digits represent the padding.This method can be used to pad messages that are any number of bits long, not necessarily a whole number of bytes long; for example, a message of 23 bits that is padded with 9 bits to fill a 32-bit block.Now that we are more familiar with RSA and modular mathematics properties, we'll explore the first interesting application that was implemented with this algorithm.

The application of RSA to verify international treaties

Let's say that Nation Alpha wants to monitor seismic data from Nation Beta to be sure that they don't experiment with any nuclear bombs in their territory. A set of sensors has been installed on the ground of Nation Beta to monitor its seismic activity, recorded, and encrypted. Then, the output data is transmitted to Nation Alpha, let's say, via a satellite.This interesting application of RSA works as follows:

- Nation Alpha, **(A)**, wants to be sure that Nation Beta, **(B)**, doesn't modify the data.
- Nation Beta wants to check the message before sending it (for spying purposes).

We name the data that's collected from the sensors **[x]**; so, the protocol works as follows:**Key generation**:

- Alpha chooses the parameters, **(N= p*q)**, as the product of two big prime numbers, and the **(e)** parameter.
- Alpha sends **(N, e)** to Beta.
- Alpha keeps the private key, **[d]**, secret.

The *protocol* for threat verification on atomic experiments is developed as follows:

- **Step 1**: A sensor located deep in the earth collects data, **[x]**, performing encryption using the private key, **[d]**:

```
x^d ≡ y (mod N)
```

- **Step 2**: Initially, both the **(x)** and **(y)** parameters are sent by the sensor to Beta to let them verify the truthfulness of the information. Beta checks the following:

```
y^e ≡ x (mod N)
```

- **Step 3**: After the positive check, Beta forwards **(x, y)** to Alpha, who can control the result of **(x)**:

```
x ≡ y^e (mod N)
```

Important Note

> **(x)** is the collected set of data from the sensor, while **[d]** is the private key of Alpha stored inside the protected software sensor that collects the data.
>
> This encryption is performed in the opposite way to how RSA usually works.

If the **y^e ≡ x (mod N)** equation is verified, Alpha can be confident that the data that's been sent from Beta is correct and they didn't modify the message or manipulate the sensor. That's because the encrypted message, **(x)**, corresponding to the cryptogram, **(y)**, can truly be generated by only those who know the private key, **[d]**.If Beta has previously attempted to manipulate the encryption inside the box that holds the sensor by changing the value of **(x)**, then it will be very difficult for Beta to get a meaningful message.As we mentioned previously, in this protocol, RSA is inverted, and the encryption is performed with the private key, **[d]**, instead of **(e)**, the public parameter, which is what normally happens.Essentially, the difficulty here for Beta in modifying the encryption is to get a meaningful number or message. Trying to modify the cryptogram, **(y)**, even if the **(x)** parameter is previously known, has the same complexity to perform the discrete logarithm (which is a hard problem to solve, as we have already seen).We visualize the process by referring to the following diagram:

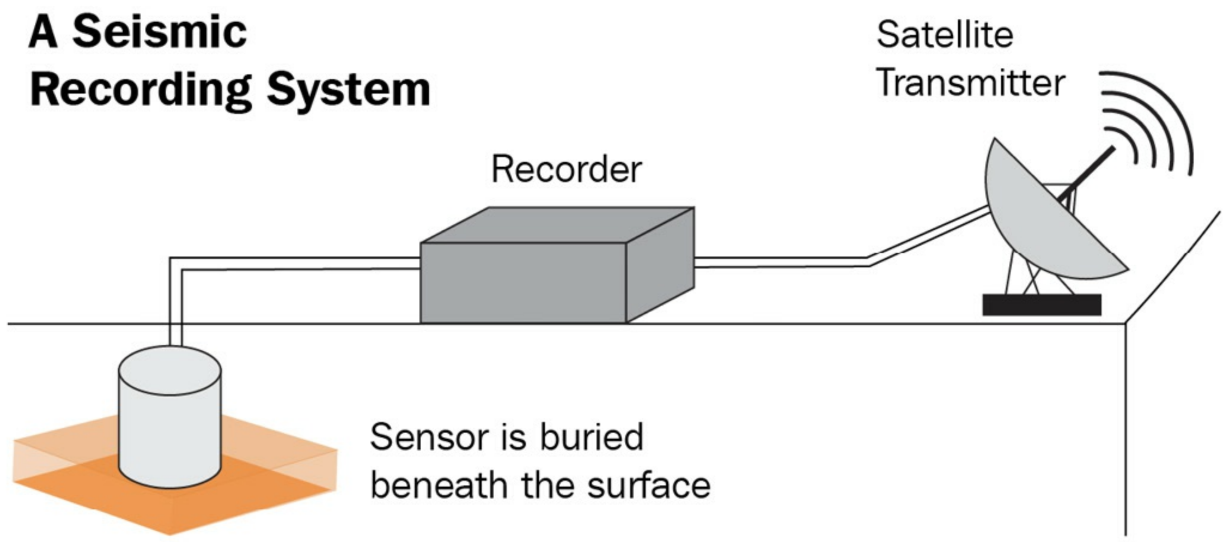

A Seismic Recording System
Satellite Transmitter
Recorder
Sensor is buried beneath the surface

Figure 3.3 – The sensor buried underground with the data transmitted via satellite

Now that we've learned how to use the international treaties that are performed with the RSA algorithm, I want to introduce a section that will be discussed later in *Chapter 6, New Algorithms in Public/Private Key Encryption*, related to unconventional attacks on the RSA algorithm and its most famous library, OpenSSL.

Unconventional attacks

I have called these algorithms *unconventional* because they have been implemented by me and not tested and published until now. We will see as we continue through this book, these *unconventional attacks* against RSA are even valid for other asymmetric encryption algorithms. These unconventional attacks, implemented between 2011 and 2014, have the scope to recover the secret message, **[M]**, without knowing Alice's decryption key, **[d]**, and the prime secret numbers, **[p]** and **[q]**, behind (**N**). I will showcase these unconventional attacks here in this section, but I will present these methods in more detail in *Chapter 6, New Algorithms in Public/Private Key Encryption*.Among these algorithms, we have some attacks that are used against RSA, but they can be transposed to most of the asymmetric algorithms covered in this chapter.A new algorithm that could break the factorization problem in the future is *NextPrime*. It derives from a *genetic algorithm* discovered by a personal dear friend who explained the mechanism to me in 2009, Gerardo Iovane. In his article, *The Set of Prime Numbers*, Gerardo Iovane describes how it is possible to get all the prime numbers starting from a simple algorithm, discarding the non-prime numbers from the pattern.

Note

> For a more accurate and thorough discussion of Gerardo Iovane's genetic algorithm, you can read *The Set of Prime Numbers* at https://arxiv.org/abs/0709.1539.

After years of work and many headaches, I have arrived at a mathematical function that represents a curve; each position on this curve represents a prime number, and between many positions lie the semiprimes (**N**) generated by two primes. This curve geometrically represents all the primes of the universe. It turns out that the position of (**N**) lies always in-between the positions on the curve of the two prime numbers, **[p]** and **[q]**, and (**N**) is almost equidistant from their positions. It's also possible to demonstrate that the prime numbers have a very clear order and are not randomly positioned and disordered as believed.The *distance* between the two prime objects, **[p]** and **[q]**, of the multiplication that determines (**N**) is equivalent to the number of primes lying between **[p]** and **[q]**. For example, the *distance prime* between 17 and 19 is zero, the distance between 1 and 100 is 25, and the distance between 10,000 and 10,500 is only 55. Right now, this algorithm is only efficient under determinate conditions: for example, when **[p]** and **[q]** are *rather close* to each other (at a polynomial distance). However, the interesting thing is that it doesn't matter how big the two primes are. I did some tests with this algorithm using primes of the caliber of 10^1,000 digits.Just to clarify how big such numbers are, you can consider that 10^80 represents the number of particles in the Universe. For instance, a semi-prime with an order of 10^1,000 digits corresponds to RSA's public key length of around 3,000 bits (one of the biggest public keys that's used in cryptography right now). It could be processed in an elapsed time of a few seconds using the NextPrime algorithm if the two primes are close to each other.At the time of writing this book, I am working on a version of the NextPrime algorithm based on a quantum computer. It could be the next generation of quantum computing factorization algorithms (similar to the Shor algorithm, which we will look at in *Chapter 8, Quantum Cryptography*).Now, let's continue to analyze how else it is possible to attack RSA. As shown in the following diagram, there are two points of attack in the algorithm: one is the factorization of (**N**), while the other is the discrete power, **[M^e]**:

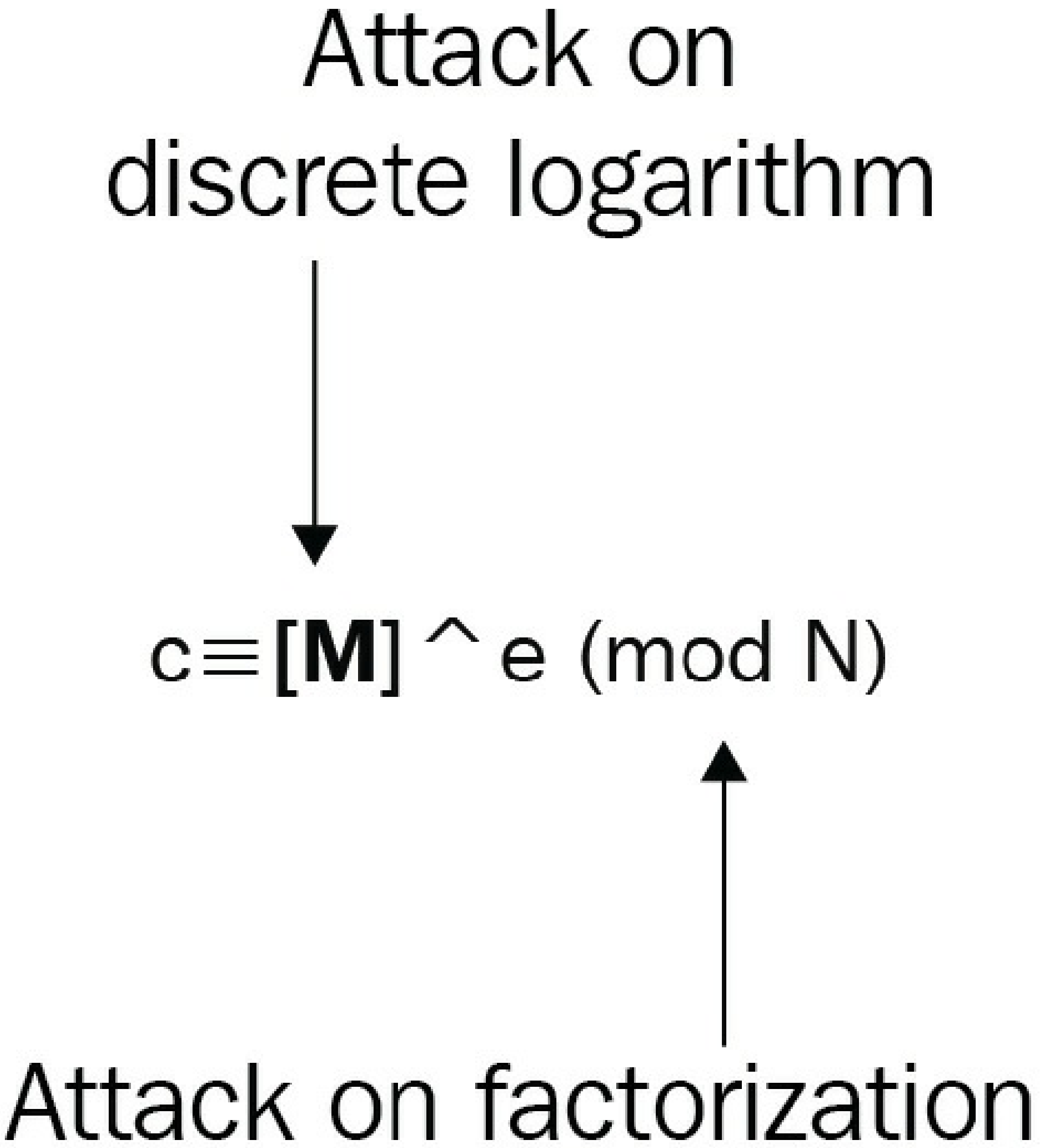


Figure 3.4 – Points where it is possible to attack RSA

Most of the *conventional* analysis of cryptologists is focused on factorizing **(N)**, as we have seen. But RSA suffers not only from the factorization problem; there is also another problem linked to the exponent, **(e)**, which we will see in *Chapter 6, New Algorithms in Public/Private Key Cryptography*.

These methods are essentially *backdoors* that can recover the message, **[M]**, without knowing the secret parameters of the sender: **[d]**, **[p]**, and **[q]**. What we will understand in *Chapter 6, New Algorithms in Public/Private Key Cryptography*, when we analyze these methods of attack, is that they are equivalent to creating a backdoor inside the RSA algorithm and its main library, OpenSSL. The RSA paradigm that we've already examined states that Bob (the sender) cannot return the message once it has been delivered. **However, if we apply those unconventional methods to break RSA, this paradigm is no longer valid.** Bob can create his "fake encryption" by himself to return the message, **[M]**, encrypted with Alice's public key, while the fake cryptogram can be decrypted by Alice using RSA's decryption stage. Is this method an unreasonable model, unrepresentative of practical situations, or does it have practical uses?I will leave the answer to this for *Chapter 6, New Algorithms in Public/Private Key Cryptography*. Now, let's explore another protocol based on the RSA implementation that has become a popular piece of software: PGP.

PGP

Pretty Good Privacy (**PGP**) is probably the most used cryptographic software in the world.PGP was implemented by Philip Zimmermann during the Cold War. Philip started planning to take his family to New Zealand because he believed that, in the event of a nuclear attack, the country, so isolated from the rest of the world, would be less impacted by atomic devastation. At some point, while planning to move to New Zealand, something changed his mind and he decided to remain in the US.To communicate with his friends, Zimmermann, who was an anti-nuclear activist, developed PGP to secure messages and files transmitted via the internet. He released the software as open source, free of charge for non-commercial use.At the time, cryptosystems larger than 40 bits were considered to be like munitions. That is to say that cryptography is still considered a military weapon. There is a requirement that you must obtain if you decide to patent a new cryptosystem, you must have authorization from the Department of Defense to publish it. This was a problem that was encountered by PGP, which never used keys smaller than 128 bits. Penalties and criminal prosecutions for violating this legal requirement are very severe, which is why Zimmermann had a *pending legal status* for many years until the American government decided to close the investigation and clear him.PGP is not a proper algorithm, but a protocol. The innovation here is just to merge asymmetric with symmetric encryption. The protocol uses an asymmetric encryption algorithm to exchange the key, and symmetric encryption to encrypt the message and obtain the ciphertext. Moreover, a digital signature is required to identify the user and avoid a MiM attack.The following are the steps of the protocol:

- **Step 1**: The key is transmitted using an asymmetric encryption algorithm (ElGamal, RSA).
- **Step 2**: The key that's been transmitted with asymmetric encryption becomes the session key for the symmetric encryption (DES (remember that we have seen that is breackble now), Triple DES, IDEA, and AES – we covered this in *Chapter 2, Introduction to Symmetric Encryption*).
- **Step 3**: A digital signature is used to identify the users (we will look at RSA digital signatures in *Chapter 4, Introducing Hash Functions and Digital Signatures*).
- **Step 4**: Decryption is performed using the symmetric key.

PGP is a good protocol for very good privacy and for securing the transmission of commercial secrets.

The ElGamal algorithm

This algorithm is an asymmetric version of the D-H algorithm. ElGamal aims to overcome the problems of MiM and the impossibility of the signatures for key ownership in D-H. Moreover, ElGamal (just like RSA) is an authentic asymmetric algorithm because it encrypts the message without previously exchanging the key.The difficulty here is commonly related to solving the discrete logarithm. As we will see later, there is also a problem related to factorization.ElGamal is the first algorithm we'll explore that presents a new element: an integer random number, **[k]**, that's chosen by the sender and kept secret. It's an important innovative element because it makes its encryption "ephemeral," in the sense that **[k]** makes the encryption function unpredictable. Moreover, we will frequently see this new element related to the zero-knowledge protocol in *Chapter 5, Introduction to Zero-Knowledge Protocols*.Let's look at the implementation of this algorithm and how it is used to transmit the secret message, **[M]**.Alice and Bob are always the two actors. Alice is the sender and Bob is the receiver.The following diagram shows the workflow of the ElGamal algorithm:

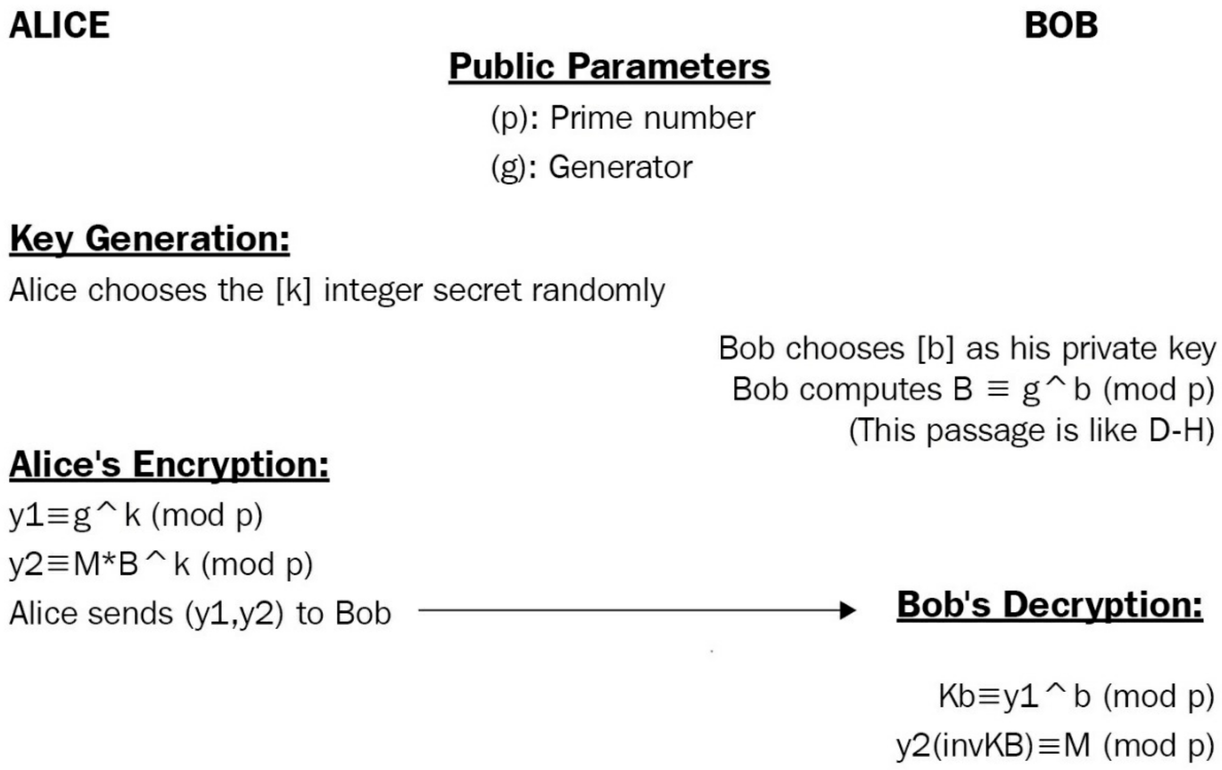


Figure 3.5 – Encryption/decryption of the ElGamal algorithm

As shown in the last step of Bob's decryption, we can see an inverse multiplication in **(mod p)**. This kind of operation is essentially a division that's performed in a finite field. So, if the inverse of **A** is **B**, we have **A*B = 1 (mod p)**. The following example shows the implementation of this inverted modular function with Mathematica.Now, having explained the algorithm, let's look at a numerical example to understand it.**Publicly defined parameters**:The public parameters are **p** (a large prime number) and **g** (a generator): `p = 200003g = 7` **Key generation**:Alice chooses a random number, **[k]**, and keeps it secret.**k = 23** (Alice's private key).Bob computes his public key, **(B)**, starting from his private key, **[b]**:**b = 2367** (Bob's private key): `B ≡ 7^2367 (mod 200003)B = 151854` **Alice's encryption**:Alice generates a secret message, **[M]**: `M = 88` Then, Alice computes **(y1, y2)**, the two public parameters that she will send to

Bob: `y1 ≡ 7^23 (mod 200003)y1 = 90914y2 ≡ 88 * 151854^ 23 (mod 20` sends the parameters to Bob (**y1** = **90914**; **y2** = **161212**).**Bob's decryption**:First, Bob computes **(Kb)** by taking **y1 = 90914**, which is elevated to his private key, **[b] = 2367**: `Kb ≡ 90914^2367 (mod 200003)Kb = 10923` Inverted **Kb** in **(mod p) [Reduce[Kb*x == 1, x, Modulus -> p]** (performed with Wolfram Mathematica): `Inverted Kb ≡ 192331 (mod 200003)` Finally, Bob can return the message, **[M]**.Bob takes **(y2)** and multiplies it by **[Inverted Kb]**, returning the message, **[M]**: `y2* InvKb ≡ M (mod 200003)` The final result is the message **[M]**: `161212 * 192331 ≡ 88 (mod 200003).` ElGamal encryption is used in the free **GNU Privacy Guard** (**GnuPG**) software. Over the years, GnuPG has gained wide popularity and become the de facto standard free software for private communication and digital signatures. GnuPG uses the most recent version of PGP to exchange cryptographic keys. For more information, you can go to the web page of this software: https://gnupg.org/software/index.xhtml.

Important Note

> As I have mentioned previously, it's assumed that the underlying problem behind the ElGamal algorithm is the discrete logarithm. This is because, as we have seen, the public parameters and keys are all defined by equations that rely on discrete logarithms.
>
> For example, **B ≡ g^b (mod p)**; **Y1 ≡ g^k (mod p)** and **Kb ≡ y1^b (mod p)** are functions related to the discrete logarithm.

Although the discrete logarithm problem is considered to be the main problem in ElGamal, we also have the factorization problem, as shown here. Let's go back to the encryption function, **(y2)**: `y2 ≡ M*B^k (mod p)` Here, you can see that there is multiplication. So, an attacker could also try to arrive at the message by factorizing **(y2)**.This will be clearer if we reduce the function: `H ≡ B^k (mod p)` Then, we have the following: `y2 ≡ M* H (mod p)` This can be rewritten like so: `M ≡ y2/H (mod p)` As you can see, **(y2)** is the product of **[M*H]**. If someone can find the factors of **(y2)**, they can probably find **[M]**.

Summary

In this chapter, we analyzed some fundamental topics surrounding asymmetric encryption. In particular, we learned how the discrete logarithm works, as well as how some of the most famous algorithms in asymmetric encryption, such as Diffie-Hellmann, RSA, and ElGamal, work. We also explored an interesting application of RSA related to exchanging sensitive data between two nations. In *Chapter 4, Introducing Hash Functions and Digital Signatures*, we will learn how to digitally sign these algorithms.Now that we have learned about the fundamentals of asymmetric encryption, it's time to analyze digital signatures. As you have already seen with PGP, all these topics are very much related to each other.

4 Hash Functions and Digital Signature

Since time immemorial, most contracts, meaning any kind of agreement between people or groups, have been written on paper and signed manually using a particular signature at the end of the document to authenticate the signatory. This was possible because, physically, the signatories were in the same place at the moment of signing. The signatories could usually trust each other because a third trustable person (a notary or legal entity) guaranteed their identities as a *super-party entity*.Nowadays, people wanting to sign contracts often don't know each other and frequently share documents to be signed via email, signing them without a trustable third party to guarantee their identities. Imagine that you are signing a contract with a third party and will be sending it via the internet. Now, consider the third party as *untrustable*, and you don't want to expose the document's contents to an unknown person via an unsecured channel such as the internet. How is it possible in this case to verify whether the signatures are correct and acceptable? Moreover, how is it possible to hide the document's content and, at the same time, allow a signature on the document?Digital signatures come to our aid to make this possible. This chapter will also show how hash functions are very useful for digitally signing encrypted documents so that anyone can identify the signers, and at the same time, the document is not exposed to prying eyes. In this chapter, we will cover the following topics:

- Hash functions
- Digital signatures with RSA and El Gamal
- Blind signatures

So, let's start by introducing hash functions and their main scopes. Then we will go deeper into categorizing digital signature algorithms.

A basic explanation of hash functions

Hash functions are widely used in cryptography for many applications. As we have already seen in *Chapter 3, Asymmetric Algorithms*, one of these applications is *blinding* an exchanged message when it is digitally signed. What does this mean? When Alice and Bob exchange a message using any asymmetric algorithm, in order to identify the sender, it commonly requires a signature. Generally, we can say that the signature is performed on the message **[M]**. For reasons we will learn later, it's discouraged to sign the secret message directly, so the sender has to first transform the message **[M]** into a function **(M')**, which everyone can see. This function is called the **hash of M**, and it will be represented in this chapter with these notations: **f(H)** or **h[M]**.We will focus more on the relations between hashes and digital signatures later on in this chapter. However, I have inserted hash functions in this chapter alongside digital signatures principally because hash functions are crucial for signing a message, even though we can find several applications related to hash functions outside of the scope of digital signatures, such as applications linked to the blockchain, like *distributed hash tables*. They also find utility in the search engine space.So, the first question we have is: what is a hash function?The answer can be found in the meaning of the word: hash=*to reduce into pieces*, which in this case refers to the contents of a message or any other information being reduced into a smaller portion. *Given an arbitrary-length message* **[M]** *as input, running a hash function* **f(H)***, we obtain an output (message digest) of a determined fixed dimension* **(M')**. If you remember the bit expansion *(Exp-function)*, seen in *Chapter 2*, *Introduction to Symmetric Encryption*, this is conceptually close to hashes. The bit expansion function works with a given input of bits that has to be expanded. We have the opposite task with hash functions, where the input dimension is bigger than the hash's bit value.We call a hash function (or simply a hash) a **unidirectional function**. We will see later that this property of being *one-way* is essential in classifying hash functions. Being a unidirectional or one-way function means that it is easy to calculate the result in one direction, but very difficult (if not impossible) to get back to the original message from the output of the function.Let's see the properties verified in a hash function:

- Given an input message **[M]**, its digest message **h(M)** can be calculated *very quickly*.
- It should be almost *impossible* (-%) to come back from the output **(M')** calculated through **h(M)** to the original message **[M]**.
- It should be computationally *intractable* to find two different input messages **[m1]** and **[m2]**, such that:

`h(m1) = h(m2)` In this case, we can say that the **f(h)** function *is strongly collision-free*.To provide an example of a hash function, if we want the message digest of the entire content of Wikipedia, it becomes a fixed-length bit as follows:

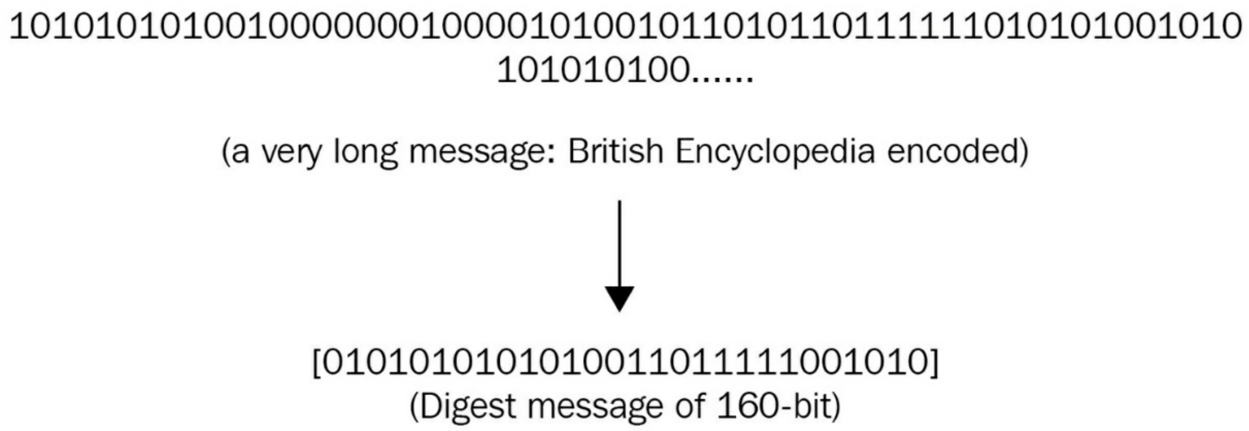


Figure 4.1 – Example of hash digest

Another excellent metaphor for hash functions could be a funnel mincer like the following, which digests plaintext as input and outputs a fixed-length hash:

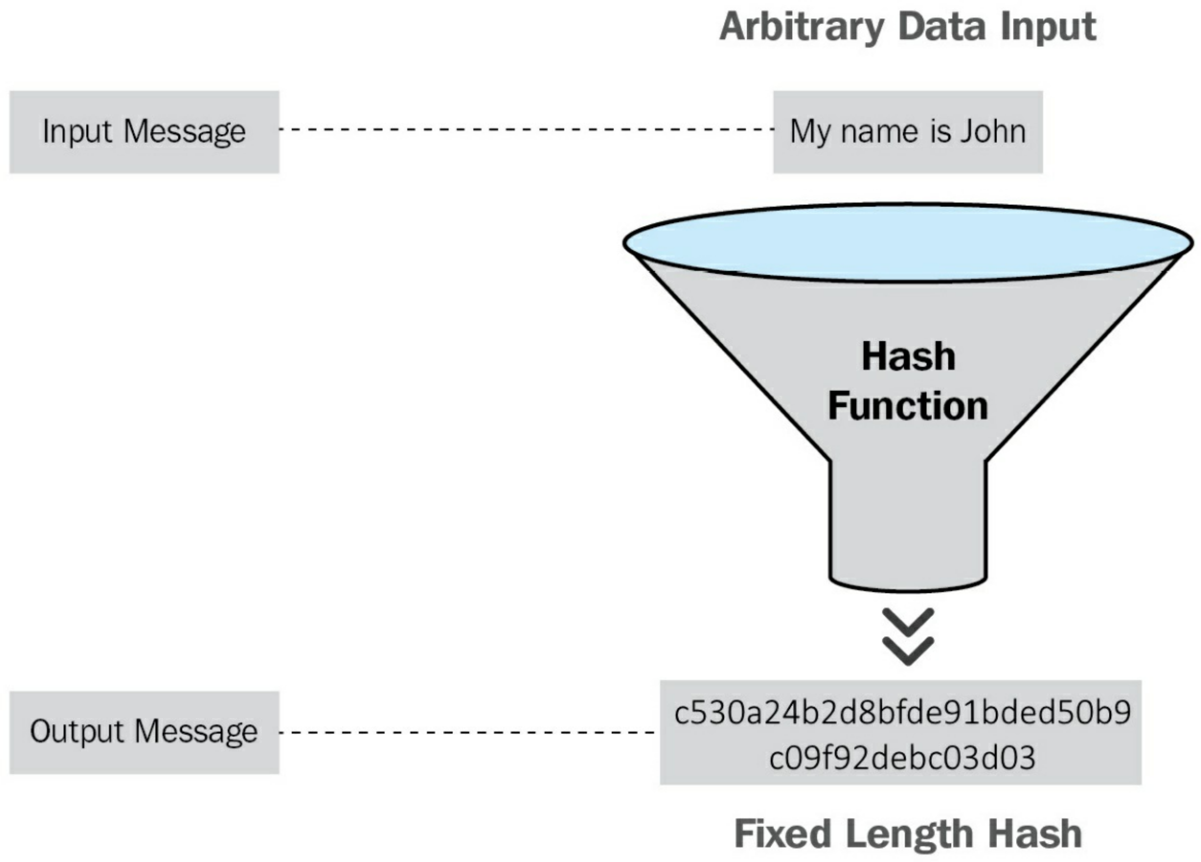


Figure 4.2 – Hash functions symbolized by a "funnel mincer"

The next question is: why and where are hash functions used?Recalling what we said at the beginning of this Chapter, in cryptography, hash functions are widely used for a range of different scopes:

- Hash functions are commonly used in asymmetric encryption to avoid exposing the original message **[M]** when collecting digital signatures on it (we will see this later in this chapter). Instead, the hash of the original message proves the identity of the transmitter using **(M')** as a surrogate.
- Hashes are used to check the integrity of a message. In this case, based on the digital hash of the original message **(M')**, the receiver can easily detect whether someone has modified the original message **[M]**. Indeed, by changing only 1 bit among the entire content of the British Encyclopedia **[M]**, for example, its hash function, **h(M)**, will have a completely different hash value **h(M')** than the previous one. This particularity of hash functions is essential because we need a strong function to verify that the original content has not been modified.

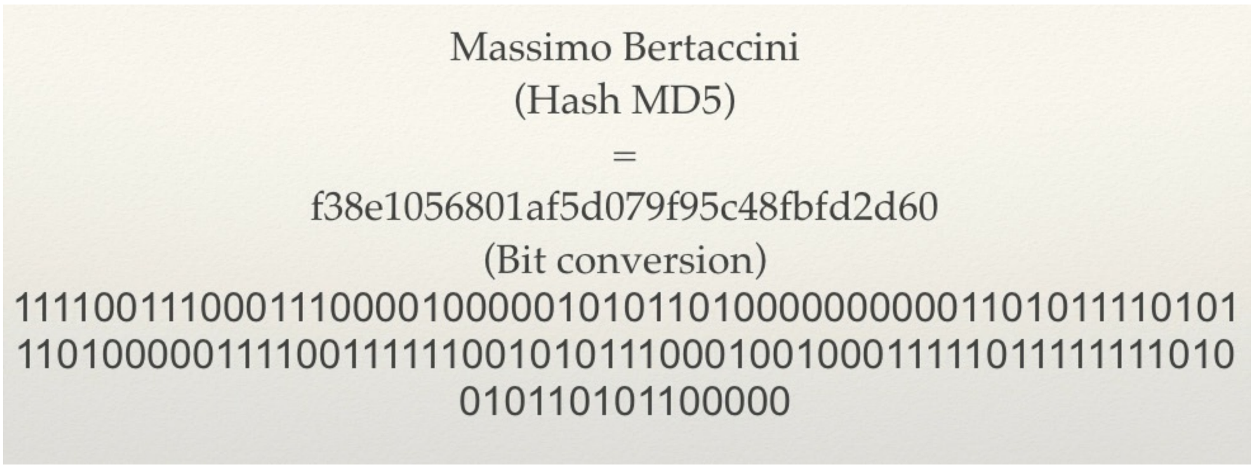
- Furthermore, hash functions are used for indexing databases. We will see these functions throughout this book. We will use them a lot in the implementation of our *Crypto Search Engine* detailed in *Chapter 9, Crypto Search Engine*, and will also see them in *Chapter 8, Quantum Cryptography*. Indeed, hash functions are candidates for being *quantum resistant*, which means that hash functions can overcome a quantum computer's attack *under certain conditions*.
- The foundational concept of performing a secure hash function is that given an output **h(M)**, it is difficult to get the original message from this output.

A function that respects such a characteristic is the *discrete logarithm*, which we have already seen in our examination of asymmetric encryption: `g^[a] ≡ y (mod p)` Where, as you may remember, given the output (**y**) and also knowing (**g**), it is very difficult to find the secret private key **[a]**.However, a function like this the discrete logarithm seems to be too slow to be considered for practical implementations, as well as weak. As you have seen above, one of the particularities of hashes is to be quick to calculate. As we have seen what hash functions are and their characteristics, now we will look at the main algorithms that implement hash functions.

Overview of the main hash algorithms

Since a hash algorithm is a particular kind of mathematical function that produces a fixed output of bits starting from a variable input, it should be collision-free, which means that it will be difficult to produce two hash functions for the same input value and vice versa.There are many types of hash algorithms, but the most common and important ones are MD5, SHA-2, and CRC32, and in this chapter, we will focus on the **Secure Hash Algorithm** (**SHA**) family.Finally, for your knowledge, there is **RIPEMD-160**, an algorithm developed by EU scientists in the early 1990s available in 160 bits and in other versions of 256 bits and 320 bits. This algorithm didn't have the same success as the SHA family, but could be the right candidate for security as it has never been broken so far. The SHA family, developed by the **National Security Agency** (**NSA**), is the object of study in this chapter. I will provide the necessary knowledge to understand and learn how the SHA family works. In particular, **SHA-256** is currently the hash function used in

Bitcoin as **Proof of Work** (**PoW**) when mining cryptocurrency. The process of creating new Bitcoins is called mining; it is done by solving an extremely complicated math problem, and this problem is based on SHA-256. At a high level, Bitcoin mining is a system in which all the transactions are devolved to the miners. Selecting one megabyte worth of transactions, miners bundle them as an input into SHA-256 and attempt to find a specific output the network accepts. The first miner to find this output receives a reward represented by a certain amount of Bitcoin. We will look at the SHA family in more detail later in this book, but now let's go on to analyze at a high level other hash functions, such as MD5.We can start by familiarizing ourselves with hash functions, experimenting with a simple example of an MD5 hash.You can try to use the **MD5 Generator** to hash your files. My name converted into hexadecimal notation with MD5 is as follows:



Massimo Bertaccini
(Hash MD5)
=
f38e1056801af5d079f95c48fbfd2d60
(Bit conversion)
11110011100011100001000001010110100000000001101011110101
11010000011110011111100101011100010010001111011111111010
010110101100000

Figure 4.3 – MD5 hash function example

As I told you before, the MD family was found to be insecure. Attacks against MD5 have been demonstrated, valid also for RIPEMD, based on differential analysis and the ability to create collisions between two different input messages.So, let's go on to explore the mathematics behind hash functions and how they are implemented.

Logic and notations to implement hash functions

This section will give you some more knowledge about the operations performed inside hash functions and their notations.Recalling Boolean logic, I will outline more symbols here than those already seen in *Chapter 2, Introduction to Symmetric Encryption*, and shore up some basic concepts. Operating in Boolean logic means, as we saw in *Chapter 2*, *Introduction to Symmetric Encryption*, performing operations on bits. For instance, taking two numbers in decimal notation and then transposing the mathematical and logical operations on their corresponding binary notations, we get the following results:

- **X** ^ **Y = AND logical conjunction** – this is a bitwise multiplication **(mod 2)**. So, the result is **1** when both the variables are **1**, and the result is **0** in all other cases:

```
X=60 —————-—>   00111100Y=240 ————-—>   11110000AND= 48————->   (
```

- **X** V **Y = OR logical disjunction** – a bitwise operation **(mod 2)** in which the result is **1** when we have at least **1** as a variable in the operation, and the result is **0** in other cases:

```
X=60 —————-—>   00111100Y=240 ——————>   11110000OR = 252 ———>   1
```

- **X Y = XOR bitwise sum (mod 2)** – we have already seen the **XOR** operation. **XOR** means that the result is **1** when bits are different, and the result is **0** in all other cases:

```
a=60 —————-—>   00111100b=240 ——————>   11110000XOR=204 ————>   1
```
operations useful for implementing hash functions are the following:

- **¬X**: This operation (the **NOT** or *inversion* operator **~**) converts the bit **1** to **0** and **0** to **1**.

So, for example, let's take the following binary string: `01010101` It will become the following: `10101010`

- **X << r**: This is the shift left bit operation. Thsis operation means to shift **(X)** bits to the left of **r** positions.

In the following example, you can see what happens when we shift to the left by **1** bit:

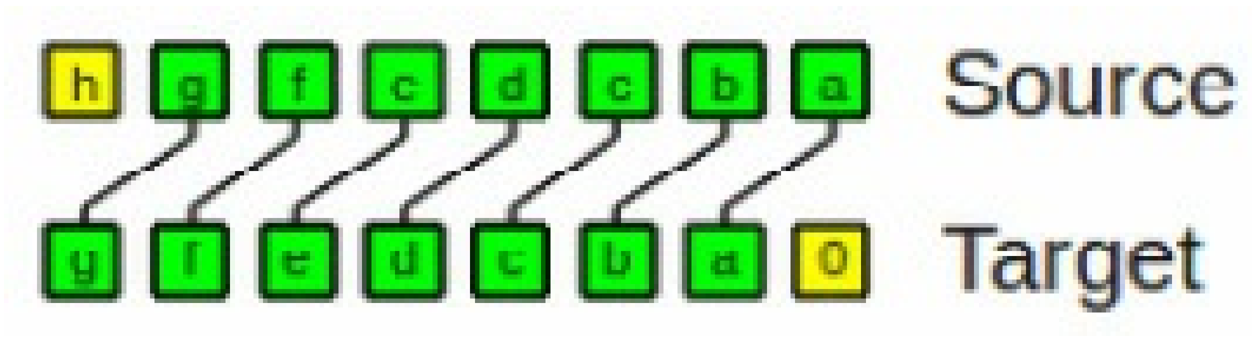


Figure 4.4 – Scheme of the shift left bit operation

So, for example, we have decimal number **23**, which is **00010111** in byte notation. If we do a shift left, we get the following:

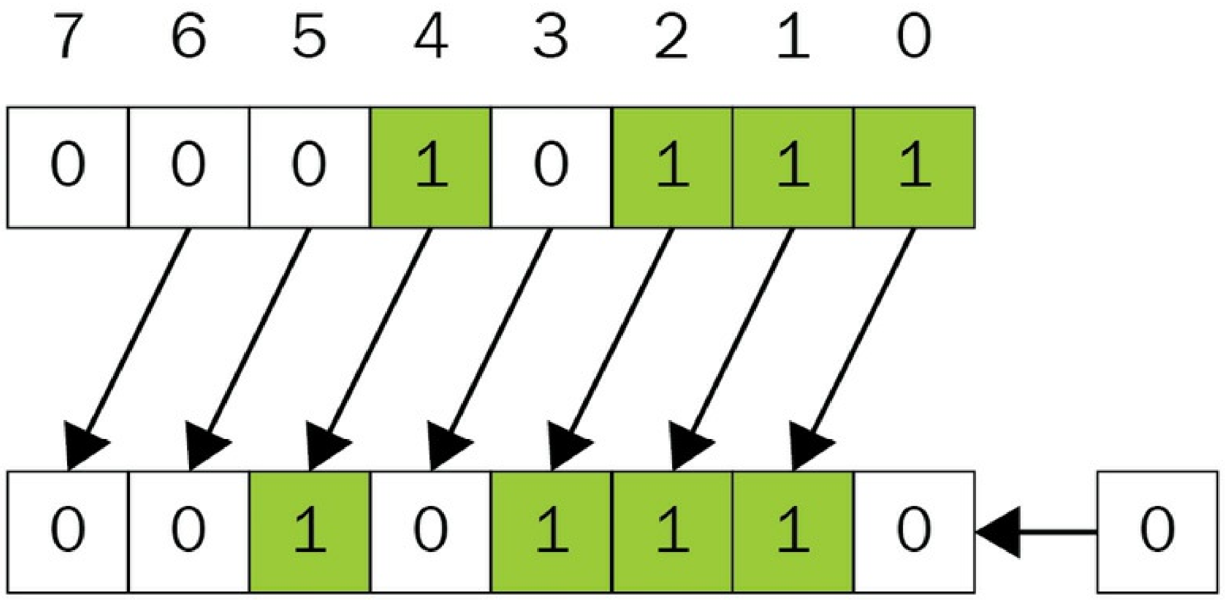


Figure 4.5 – Example of the shift left bit operation (1 position)

The result of the operation after the shift left bit is **(00101110)2** = **46** in decimal notation.

- **X** >> **r**: This is the shift right bit operation. This is a similar operation as the previous, but it shifts the bits to the right.

The following diagram shows the scheme of the shift right bit operation:

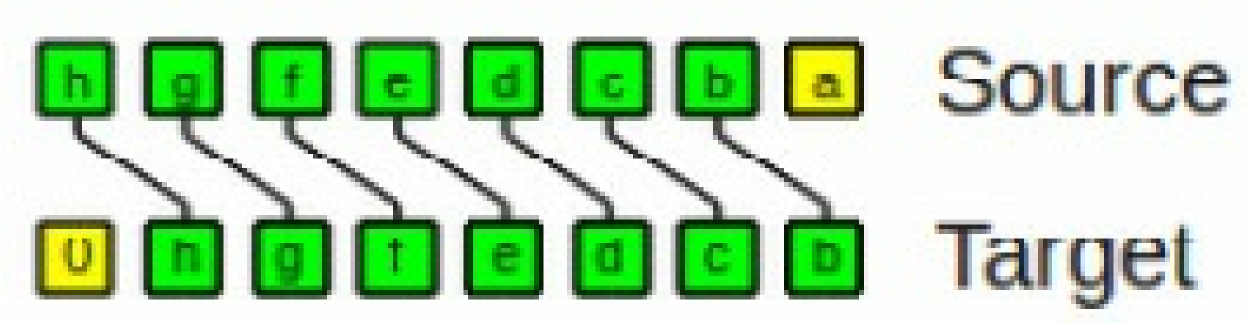


Figure 4.6 – Scheme of the shift right bit operation (1 position)

As before, let's consider this for the decimal number **23**, which is **00010111** in byte notation. If we do a shift right bit operation, we will get the decimal number **11** as the result, as you can see in the following example:

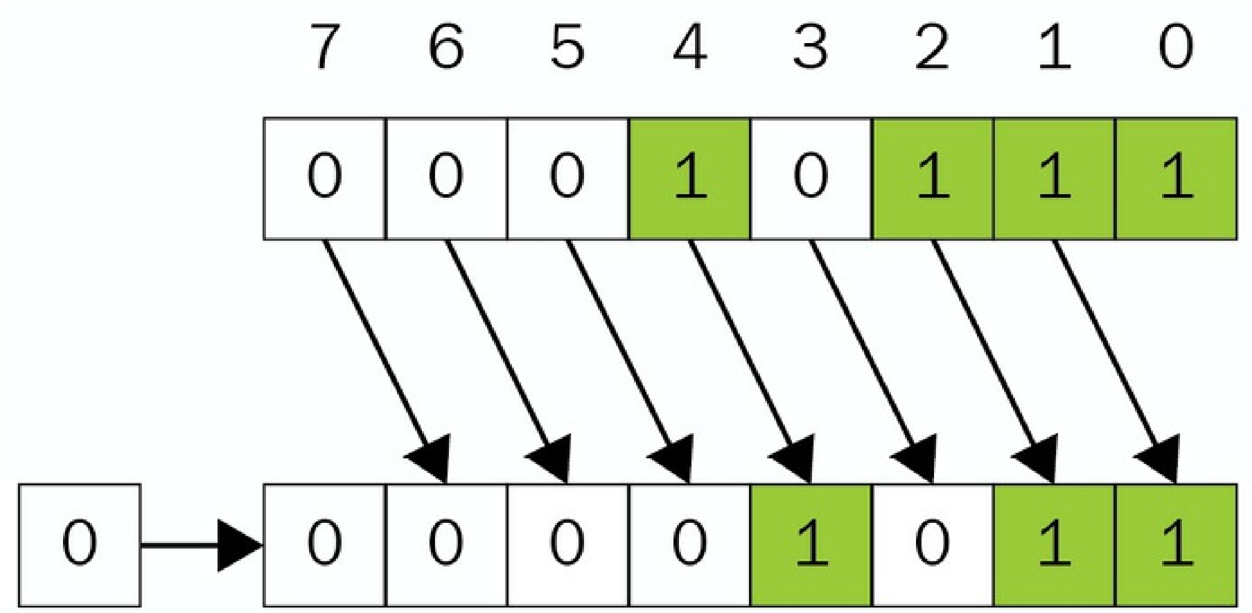


Figure 4.7 – Example of the shift right bit operation (1 position)

- **Xr**: This is left bit rotation. This operator (also represented as <<<) means a circular rotation of bits, similar to shift, but with the key difference here that the initial part becomes the final part. It's used in the SHA-1 algorithm to rotate the variables A and B, respectively, by **5** and **30** positions (**A<<<5**; **B<<<30**) as will be further explained in the following section.

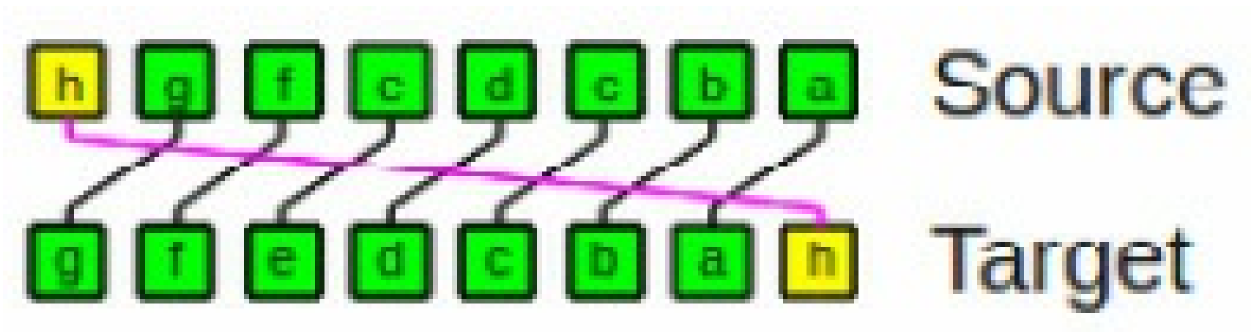


Figure 4.8 – Scheme of left bit rotation (1 position)

If we apply left bit rotation to our example of the number **23** as expressed in binary notation, we get the following:

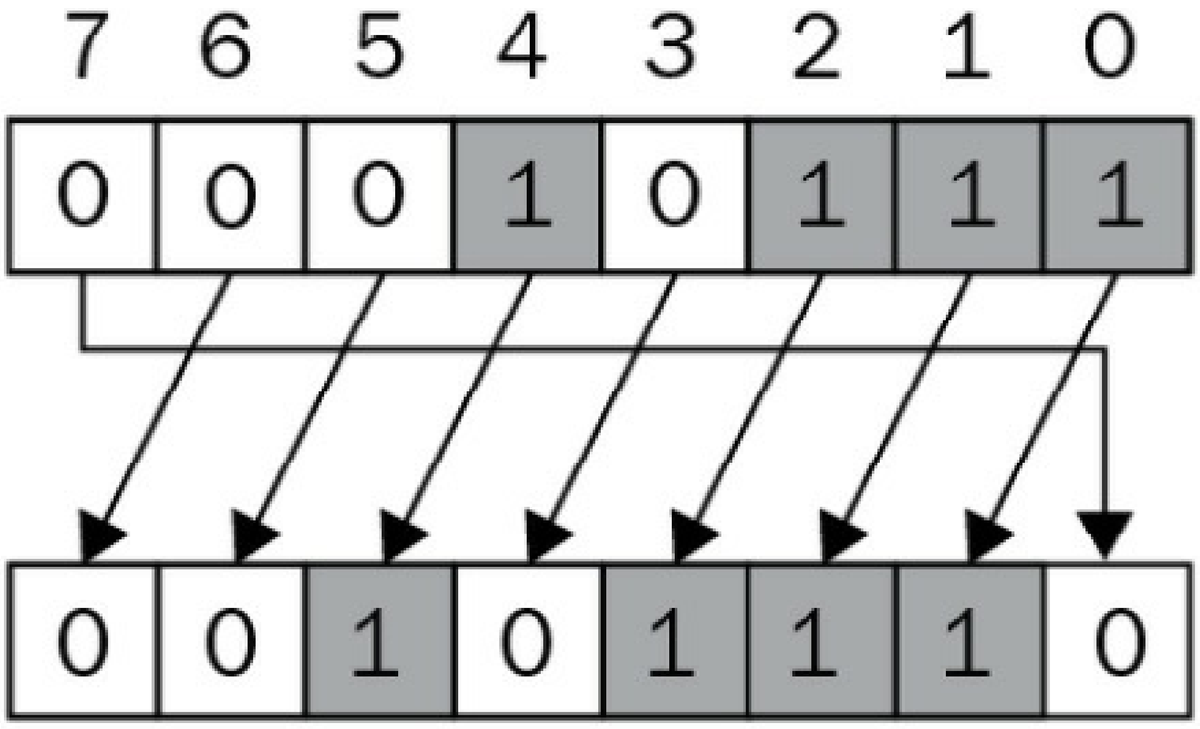


Figure 4.9 – Example of left bit rotation

In this case, the result of the left bit rotation operation is the same as the shift left bit operation: $\mathbf{(00101110)_2 = 46}$. But, if we perform a right bit rotation, we will get the following:

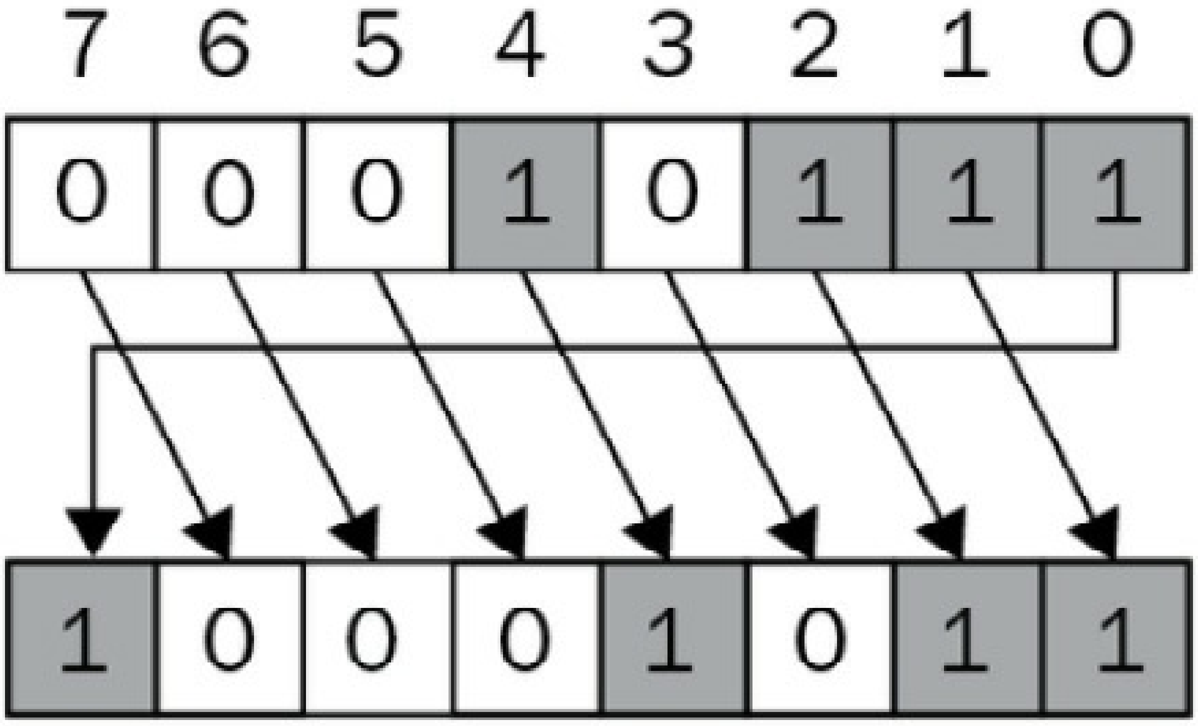


Figure 4.10 – Example of right bit rotation

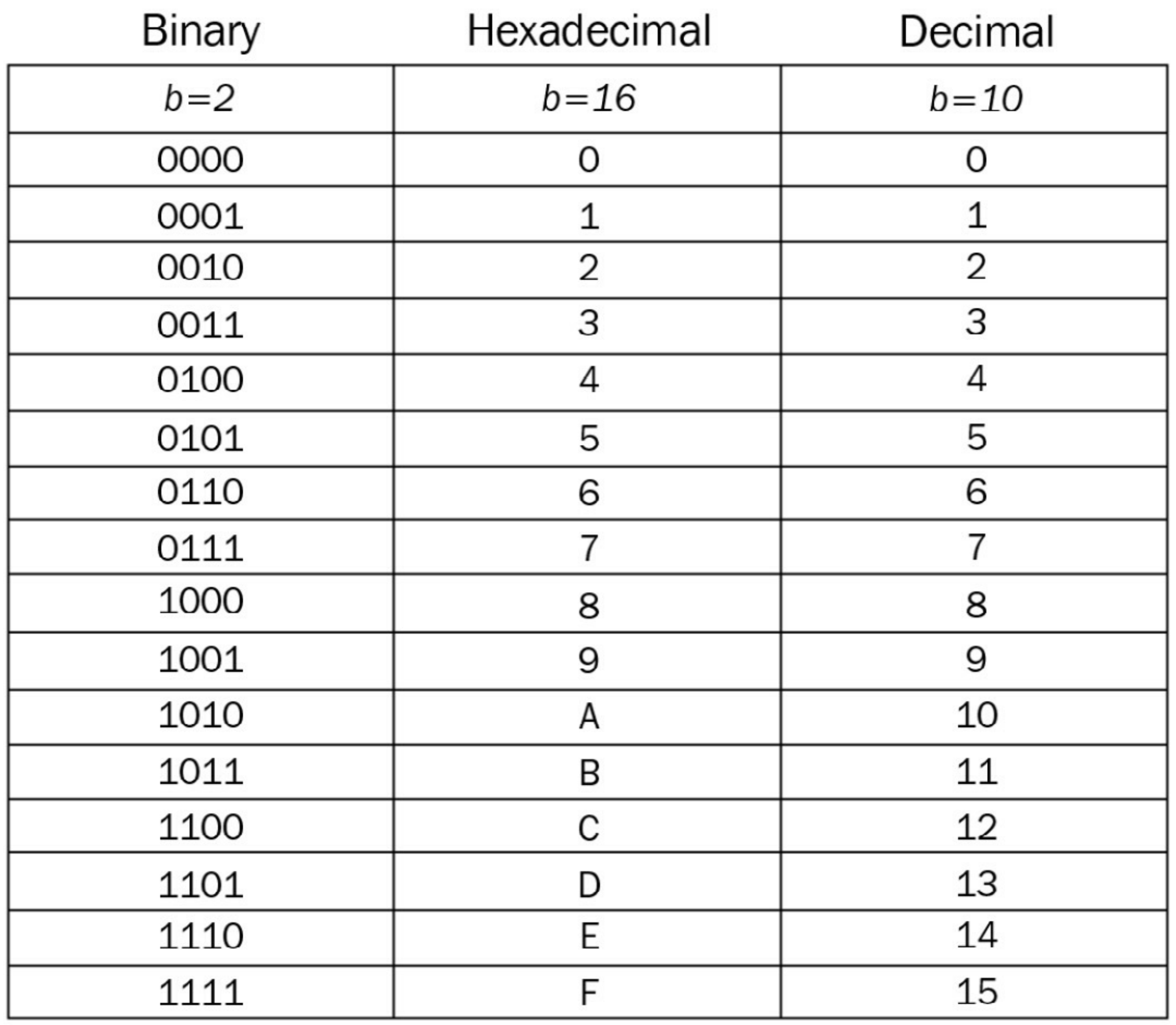
In this case, the result of the operation is **$(10001011)_2$** = **139** expressed in decimal notation.

- This operator represents the modular **sum (X+Y) (mod 2^32)** and is used in SHA to represent this operation, as you will see later in our examination of the SHA-1 algorithm (*Figure 4.14*).

After having analyzed the logic operators used to perform hash functions, two more issues have to be considered in order to implement hash functions:

- In SHA, for example, there are some constants (**Kt**) that go from **0** to **n** (we will look at this in the following section).
- The notations are generally expressed in hexadecimal.

Let's now see how the hexadecimal system works.It uses the binary numbers for 0 to 9, consisting of 4 bits per number, for example: `0= 00001= 00012= 0010………9= 1001` Then, from 10 to 16 we have 6 capital letters, **A**, **B**, **C**, **D**, **E**, **F**, to reach a total of 16 (hex) numbers: `A= 1010B= 1011……….F= 1111` For a better and clearer understanding, in the following figure, you can see a comparison between the binary, hexadecimal, and decimal systems:



Binary	Hexadecimal	Decimal
b=2	*b*=16	*b*=10
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Figure 4.11 – Comparison between binary, hexadecimal, and decimal systems

The hexadecimal system is used to represent bytes, where 1 byte is an 8-digit binary number.For example, 1 byte (8 bits) is expressed as **(1111 0010)2 = [F2]16 = 242**.Now that we have the instruments to define a hash function, let's go ahead and implement SHA-1, which is the simplest model in the SHA family.

Explanation of the SHA-1 algorithm

Secure Hash Algorithm 1 (**SHA-1**) was designed by the NSA. Since 2005, it has been considered insecure and was replaced by its successors, SHA-2 and SHA-3. In this section, we will examine SHA-1 simply as a case study to better understand how hash functions are implemented.SHA-1 returns a 160-bit output based on an iterative procedure. This concept will become clearer in the next few lines. As for other hash functions, the message **[m]** made of variable bit input, is broken into 512-bit fixed-length blocks: **m= [m1,m2,m3,....ml]**.In the last part of this section, you will see how an input message **[m]** of 2,800 bits will be transformed into blocks **[m1,m2,.....ml]** of 512 bits each.The blocks are elaborated through a compression function **f(H)** (it will be better analyzed later in *Step 3* of the algorithm) that combines the current block with the result obtained in the previous round. There are four rounds and they correspond to the variable (**t**), whose range is divided into four t-rounds as shown in *Figure 4.12*, each one made of 20 steps (for a total of 80 steps). Each iteration can be seen as a counter that runs along the values of each range made of 20 values. As you can see from *Figures 4.12* and *4.13*, each iteration uses the constants (**Kt**) and the operations **ft (B,C,D)** of the corresponding round. Each round updates the sub-registers (A,B,C,D,E) after the other. At the end of the 4th round, when t = 79, the sub-registers (A,B,C,D,E) are added to the sub-registers (H0, H1, H2, H3, H4) to perform the 160 bits final hash value.Let's now examine the issue of constants in SHA-1.Constants are fixed numbers expressed in hexadecimal defined with particular criteria. An important criterion adopted to choose the constants in SHA is to avoid collisions. Collisions happen when a hash function gives the same result for two different blocks starting from different constants. So, even though someone might think it would be a good idea to change the values of the constants, don't try to change them because that could cause a collision problem. In SHA-1, for example, the given constants are as follows:

$$Kt = \begin{cases} \text{5A827999} & \text{if } 0 \leq t \geq 19 \\ \text{6ED9EBA1} & \text{if } 20 \leq t \geq 39 \\ \text{8F1BBCDC} & \text{if } 40 \leq t \geq 59 \\ \text{CA62C1D6} & \text{if } 60 \leq t \geq 79 \end{cases}$$

Figure 4.12 – The Kt constants in SHA-1

Where, in different ranges of (**t**), different values correspond with the constants (**Kt**), as you see in the preceding figure.Besides the constants, we have the function **ft (B,C,D)** defined as follows:

$$f_t\ (B, C, D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq t \geq 19 \\ B \oplus C \oplus D & \text{if } 20 \leq t \geq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq t \geq 59 \\ B \oplus C \oplus D & \text{if } 60 \leq t \geq 79 \end{cases}$$

Figure 4.13 – The ft function

The first initial register of SHA-1 is X0, a **160**-bit hash function generated by five sub-registers (**H0**, **H1**, **H2**, **H3**, **H4**) consisting of **32** bits each. We will see the initialization of these sub-registers in *Step 2* using constants expressed in hexadecimal numbers.Now let's explain the SHA-1 algorithm by dividing the process into four steps to obtain the final hash value of **160** bits:

- **Step 1** – Starting with a message **[m]**, operate a concatenation of bits such that:

`y= m1 || m2 || m3 || … ||mL` Where the ‖ symbol stands for the concatenation of bits expressed by each block of message **[ml]** consisting of **512** bits.

- **Step 2** – Initialization of the sub registers: **H0 = 67452301**, **H1 = EFCDAB89**, **H2 = 98BADCFE**, **H3 = 10325476**, **H4 = C3D2E1F0**.

You may notice that these constants are expressed in hexadecimal notation.

They were chosen by the NSA, the designers of this algorithm.

- **Step 3** – For **j = 0, 1, … ,L-1**, execute the following instructions:

a) **mi = W0 ‖ W1……….. ‖W15**,where each **(Wj)** consists of **32** bits.b) For **t = 16** to **79**, put: `Wt = (Wt-3 Wt-8 Wt-14 Wt-16) 1` c) At the beginning, we put: `A = H0, B = H1, C = H2, D = H3, E = H4` Each variable (A,B,C,D,E) is of 32 bits length for a total length of the sub-register of 160 bits.d) For 80 iterations, where **0** ≤ **t** ≤**79**, execute the following steps in succession: `T = (A 5) + ft((B,C, D) +E + Wt + KtE = D, D = C, C = (`
The sub-registers (A,B,C,D,E) are added to the sub-registers (H0, H1, H2, H3,
H4): `H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 +`

- **Step 4** – Take the following as output:

`H0 ‖ H1 ‖ H2 ‖ H3 ‖ H4.` This is the hash value of **160** bits.Here you can see a scheme of SHA-1 (the sub-registers are **A**, **B**, **C**, **D**, **E**):

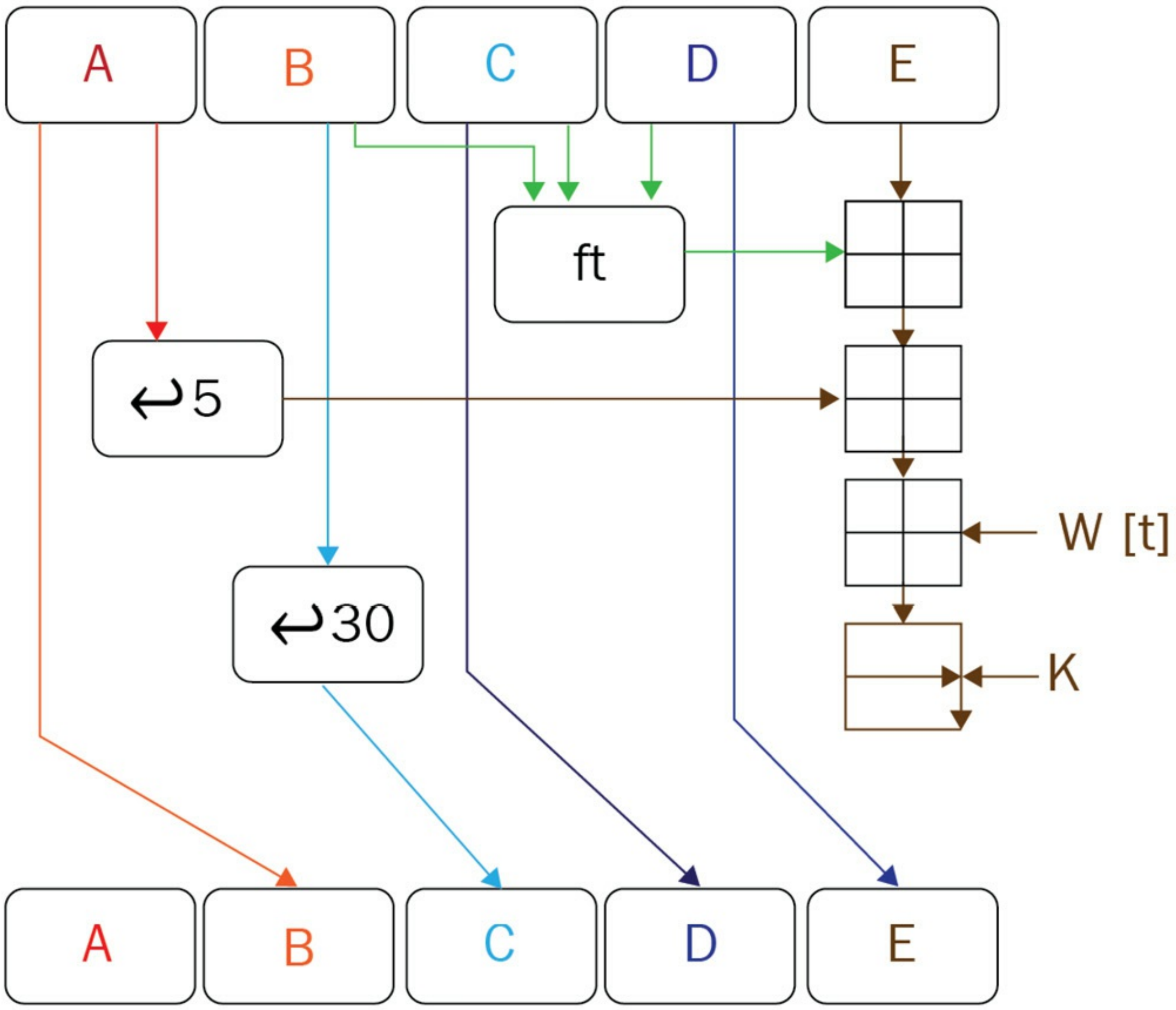


Figure 4.14 – SHA-1 scheme of the operations in each sub-register

Important Note

Remember from the previous section on the logic and notations that:

Xr is left bit rotation, also represented as <<<, and means a circular rotation of bits. So, in the case of A 5, it means that bits rotate 5 positions to the left, while in the case of B 30 bits, bits rotate 30 positions to the left.

This operator represents the modular **sum (X+Y) (mod 2^32)**.

Notes and example on SHA-1

In this section, we will analyze the SHA-1 algorithm in a little more detail.Since *Steps 1* and *2* are just message initialization and sub-register initialization, the algorithm's core is in *Step 3*, where you can see a series of mathematical operations consisting of bit concatenation, **XOR**, bit shift, bit transposition, and bit addition.Finally, *Step 4* reduces the output to **160** bits just because each sub-register **H0**, **H1**, **H2**, **H3**, and **H4** is **32** bits. In *Step 1*, we mentioned that the minimum block message provided in the input has to be **512** bits. The message **[m]** could be divided into blocks of **512** bits, and if the original message is shorter than **512** bits, we have to apply a padding operation, involving the addition of bits to complete the block.SHA-1 is used to compute a message digest of **160**-bit length for an arbitrary message that is provided as input. The input message is a bit string, so the length of the message is the number of bits that make up the message (that is, an empty message has length **0**). If the number of bits in a message is a multiple of **8** (a byte), for compactness, we can represent the message in hexadecimal. Conversely, if the message is not a multiple of a byte, then we have to apply padding. The purpose of the padding is to make the total length of a padded message a multiple of **512**. As SHA-1 sequentially processes blocks of **512** bits when computing the message digest, the trick to getting a strong message digest is that the hash function has to provide the best grade of confusion/diffusion on the bits involved in the iterative operations. The process of padding consists of the following sequence of passages:

- SHA-1 starts by taking the original message and appends **1** bit followed by a sequence of **0** bits.
- The **0** bits are added to ensure that the length of the new message is a multiple of **512** bits.
- For this scope, the new message will have a final length of **n*512**.

For example, if the original message has a length of **2800** bits, it will be padded with **1** bit at the end followed by **207** bits. So, we will have **2800 + 1 + 207 = 3008** bits. Then to make the result of the padding a multiple of 512, using the division algorithm, notice that **3008 = 5 * 512 + 448**, so to get to a multiple of **512**, we pad the message with 64 zeros. So, we finally obtain 3008 + 64 = 3072, which is the number of bits of the padded message, divisible by 512 (bits per block).

Example of one block encoded with SHA- 1

Let's understand this with the help of a practical example:

- **Step 1 – Padding the message**:

Suppose we want to encode the message **abc** using SHA-1, which in binary system is expressed as: `abc = 01100001 01100010 01100011` In hex, the string is: `abc = 616263` As you can see in the following figure, the message is padded by appending 1, followed by enough 0s until the length of the message becomes 448 bits. Since the message **abc** is 24 bits in length, 423 further bits are added. The length of the message represented by 64 bits is then added to the end, producing a message that is 512 bits long:

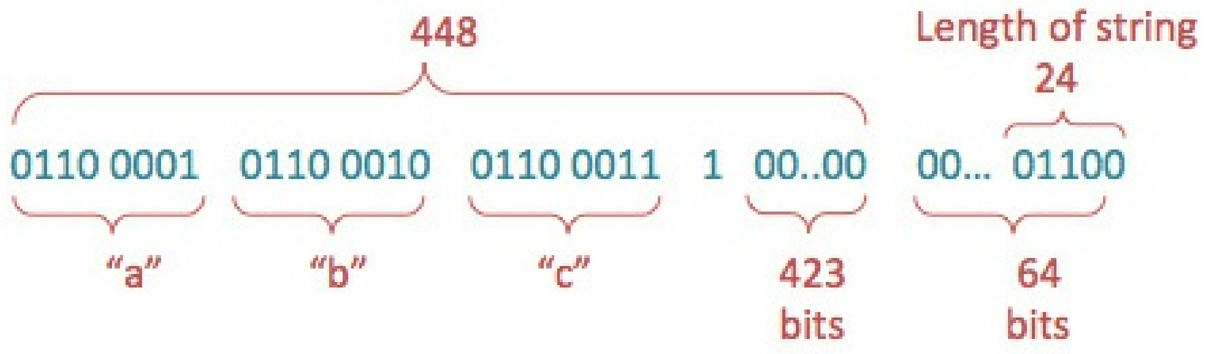


Figure 4.15 – Padding of the message in SHA-1

- **Step 2 – Initialization of the sub-registers**:

The initial hash value for the sub-registers H0, H1, H2, H3, and H4 will be:

```
H[0] = 67452301H[1] = EFCDAB89H[2] = 98BADCFEH[3] = 10325476H[4]
```

- **Step 3 – Block contents**:

```
W[0] = 61626380 W[1] = 00000000 W[2] = 00000000 W[3] = 00000000 W
```

on the sub-

registers: `A          B          C          D          Et = 0: 011`

of the sub-registers: `H[0] = 67452301 + 42541B35 = A9993E36 H[1] = EFCDAB89 + 5`

- **Step 4** – **Result**:

After performing the four rounds, the final message digest of the string **abc** of 160-bit hash is: `A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D`

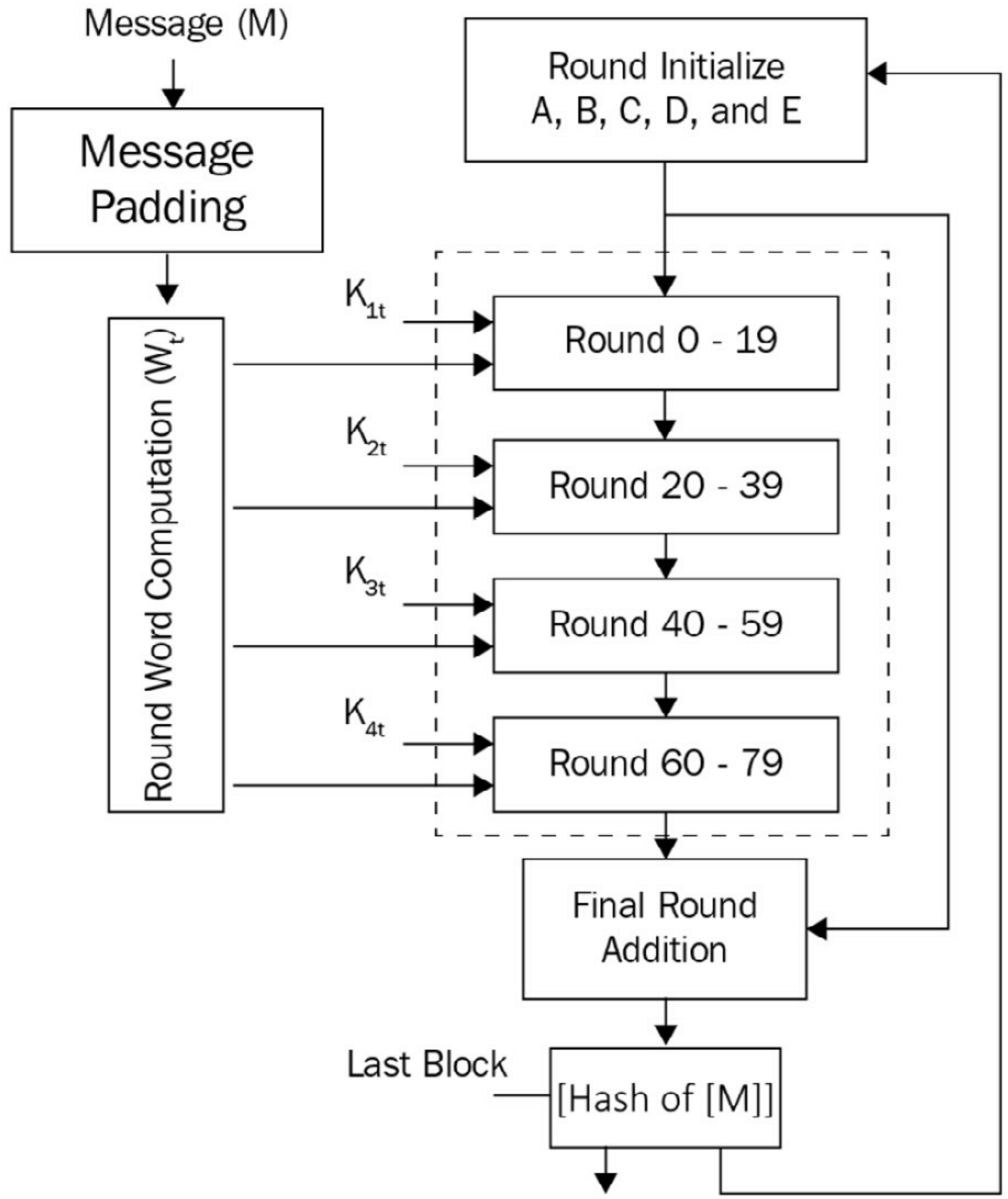


Figure 4.16 – A complete round in SHA-1

Now that we have learned about hash functions, we are prepared to explore digital signatures.

Authentication and digital signatures

In cryptography, the **authentication** problem is one of the most interesting and difficult problems to solve. Authentication is one of the most sensitive functions (as well as the most used) for the procedure of access control.Authentication is based on three methods:

- On something that *only the user knows* (for example, a password)
- On something that *only the user holds* (a smart card, device, or a token)
- On something that *characterizes the user* (for example, fingerprints, an iris scan, and in general biometric characteristics of a person)

In addition to these three methods, there is one more method involving something that *only the user holds, related to something that uniquely characterizes them*: **brain waves.** For example, when you look at a picture or think about it, the brain waves activated by your brain are unique, different from any other brain waves that characterize another person.This invention is related to the authentication between human/computer. In other words, with Brain Password, it is possible to unlock devices such as computers and telephones and get access to web applications. I will introduce brain passwords in *Chapter 6, New Algorithms in Public/Private Key Cryptography*.In the next section, we will analyze an authentication method based on digital signatures. Keep in mind (as we will see later on) that there are similar methods of authentication based, for example, on zero knowledge, which will be covered further in *Chapter 5, Introduction to Zero-Knowledge Protocols*.Let's see how a method of authentication based on a digital signature over public/private key encryption works.sLet's consider an example where Alice wants to transmit a message to Bob. She knows Bob's public key (as in RSA), so Alice encrypts the message **[M]** with RSA and sends it to Bob.Let's look at some of the common problems faced when Alice transmits the message:

- How can Bob be sure that this message comes from Alice (*authentication problem*)?
- After Bob has received the message, Alice could always deny that she is the transmitter (non-*repudiation*).
- Another possible issue is that the message **[M]** could be manipulated by

someone who intercepted it (a **Man-in-the-Middle** (**MiM**) attack) and subsequently, the attacker could change part of the content. This is a case of *integrity loss* of the message.
- Finally, here's another consideration (and probably the first time you'll see it in a textbook): the message could be intercepted and *spied* on by the attacker, who recovers the content but decides not to modify it. How can the receiver and the transmitter be sure that this doesn't happen? For example, how can you be sure that your telecommunication provider or the cloud that hosts your data don't spy on your messages? I refer to this as the spying problem, and you will see that there is a solution to discover and avoid also this problem.

To answer the last question, you could rightly say that since the message **[M]** is encrypted, it is difficult – if not impossible – for the attacker to recover the secret message without knowing the key.I will demonstrate that (under some conditions) it is possible to spy on an encrypted message, even if the attacker doesn't know the secret key for decryption.We will discuss problems *1*, *2*, and *3* in this chapter. As for problem *4*, I will explain an attack method for spying and the relative way to repair this problem in *Chapter 6, New Algorithms in Public/Private Key Cryptography*. Where I explain many of my Algorithms and some issues found during my carrier with the relative solutions of defence,A digital signature is a proof that the sender of the message **[M]** instructs the receiver to do the following:

- Prove their identity
- Prove that the message **[M]** has not been manipulated
- Provide for the non-repudiation of the message

Let's see how digital signatures can help us to avoid all these problems and also how it is possible to sign a message based on the type of algorithm and the different ways of using signatures. We will first look at RSA digital signatures, and will then analyze the other methods of signatures in public-private key algorithms.

RSA digital signatures

As I have told you before, it's possible to sign a message in different ways;

we will now explore how to mathematically sign one and how anyone can verify a signature.Recalling the RSA algorithm from *Chapter 3, Asymmetric Encryption*, we know that the process of encryption for Bob is as follows:Perform encryption based on the public key of Alice **(Na)** and define the secret elements within the square brackets **[M]**: `[M]^e ≡ c (mod Na)` **(Na)** is the public key of the receiver (Alice). Therefore, Bob will encrypt the message **[M]** with Alice's public key, and Alice will decrypt **[M]** with her private key **[da]** by performing the following operation: `c^[da] ≡ [M] (mod Na)` Where the parameter **[da]** is the private key of Alice, given by the operation: `INV (e) ≡ [da] mod(p-1)(q-1)` The process for the digital signing and verification of Bob's identity is as follows:

- **Step 1**: Bob chooses two big prime numbers, **[p1, q1]**, and keeps them secret.
- **Step 2**: Bob calculates **Nb = [p1] * [q1]** and publishes **(Nb)** as his public key.
- **Step 3**: Bob calculates his private key **[db]** such that:

```
INV (e) ≡ [db] mod (p1-1)(q1-1)
```

- **Step 4**: Bob performs the signature:

`S ≡ [M]^[db] (mod Nb)` Where **(S)** and **(Nb)** are public and **[db]** is secret. What about **[M]**?**[M]**, the message, is supposed to be secret and shared only between Bob and Alice. However, a digital signature should be verified by everyone. We will deal with this issue later on. Now let's verify the signature (**S**).

- **Step 5**: Verifying the signature:

Signature verification is the inverse process of the above. Alice (or anyone) can verify the following: `S^e ≡ [M] (mod Nb)` If the preceding equation is verified, then Alice accepts the message **[M]**.Before analyzing the issue of **[M]**, let's understand this algorithm with the help of a numerical example.**Numerical example**:Recalling the RSA example from *Chapter 3, Asymmetric Encryption*, we have the following numerical parameters given by the RSA algorithm: `[M] = 88e = 9007` Alice's parameters were as follows: `[p] = 101[q] = 67Na = 6767` Let's perform all of the steps of RSA to show the comprehensive process of digitally signing the message:

- **Step 1** – Bob's encryption is as follows:

`[M] ^e ≡ c (mod N)88^9007 ≡ 6621 (mod 6767)c = 6621`

- **Step 2** – Bob's signature **(S)** for the message **[M]** will be generated as follows:

Bob chooses two prime numbers, **[p, q]**: `[p1] = 211[q1] = 113Nb= 211*113 = 238439007 * [db] ≡ 1 (mod (2` the modular equation for **[db]**, we have the following: `[db] = 9103` Now, Bob can sign the message: `[M]^[db] ≡ S (mod Nb)88^9103 ≡ 19354 (mod 23843)S = 19354` sends to Alice the pair **(c, S) = (6621, 19354)**.

- **Step 3** – Alice's decryption will be as follows:

`c^[da] ≡ [M] (mod Na)` Alice calculates **[da]**: `9007 * [da] ≡ 1 (mod (101-1)*(67-1))[da] = 3943` Alice decrypts the cryptogram **(c)** and obtains the message **[M]**: `c^[da] ≡ [M] (mod Na)6621^3943 ≡ 88 (mod 6767)`

- **Step 4** – Verification of Bob's identity:

If the signature **(S)** elevated to the parameter **(e)** gives the message **[M]**, then Alice can be sure that the message was truly sent by Bob: `S^e ≡ [M] (mod Nb)19354^9007 = 88 (mod 23843)` Indeed, Alice obtains the message **M=88**.

Important Note

I hope someone has noticed that the message **[M]** can be verified by anyone, and not only by Alice, because **(S, e, Nb)** are public parameters. So, if Bob uses the original message **[M]**, instead of its hash **h[M]** to gain the signature **(S)** (see above for the encryption procedure), everyone who knows Bob's public key could easily recover the message **[M]**!

It's just a matter of solving the following equation to recover the secret message **[M]**: `S^e ≡ x (mod Nb)` Where the parameters **(S, e, Nb)** are all known.In this case, *hash functions* come to our aid. Performing the hash function **h[M] = (m)**, Bob will send the couple **(c, S)** , signing **(m)** instead of **[M]**.Alice already got the encrypted message **[M]**, so only Alice can verify it: `h[M] = m` If it is **TRUE**, then Bob's identity will be verified by Alice; if it is **FALSE**, Bob's claim of identity will be refused.

Why do digital signatures work?

If anyone else who isn't Bob tries to use this signature, they will struggle with the *discrete logarithm* problem. Indeed, generating **[db]** in the following equation is a hard problem for the attacker to solve: `m^[db] ≡ S (mod Nb)` Also, even if **(m, S)** are known by the attacker, it is still very difficult to calculate **[db]**. In this case, we are dealing with a discrete logarithm problem like what we saw in *Chapter 3, Asymmetric Algorithms*.Let's try to understand what happens if Eve (an attacker) tries to modify the signature with the help of an example:Eve (the attacker) exchanges **[db]** with **(de)**, computing a fake digital signature **(S')**: `m^(de) ≡ S' (mod Nb)` If Eve is able to trick Alice to accept the signature **(S')**, then Eve can make an MiM attack by pretending to be Bob, substituting **(S')** with the real signature **(S)**.But when Alice verifies the signature **(S')**, she recognizes that it doesn't correspond to the correct signature performed by Bob because the hash of the message is **(m')**, not **(m)**: `(S')^e ≡ m' (mod Nb)m' m` So, Alice refuses the digital signature and will not open any message coming from this fake address.That is why cryptographers also need to pay a lot of attention to the collisions between hashes.Now that we have got a different result, that is, **(m')** instead of **(m)**, Alice understands there is a problem and doesn't accept the message **[M]**.This is the scope of the signature **(S)**.The preceding attack is a simple trick, but there are some more intelligent and sneaky attacks that we will see later on, in *Chapter 6, New Algorithms in Public/Private Key Cryptography*, when I will explain unconventional attacks.

Digital signatures with the ElGamal algorithm

ElGamal is a public-private key algorithm, as we saw in *Chapter 3, Asymmetric Algorithms*, based on the *Diffie-Hellman key exchange*. In ElGamal, we have different ways of signing the message than RSA, but all are equally valid.Recalling the ElGamal encryption technique, we have the following elements:

- **(g, p)**: Public parameters
- **[k]**: Alice's private key
- **[M]**: The secret message
- **B ≡ g^[b] (mod p)**: Bob's public key
- **A ≡ g^[a] (mod p)**: Alice's public key

Alice's encryption: `y1 ≡ g^[k] (mod p)y2 ≡ [M]*B^[k] (mod p)`

Note

> Remember that the elements inside the square brackets indicate secret parameters; all the others are public.

Now, if Alice wants to add her digital signature to the message, she will make a *hash of the message*, **h[M]**, to protect the message **[M]**, and will then transmit the result to Bob in order to prove her identity.To sign the message **[M]**, Alice first has to generate the hash of the message **h[M]**: `h[M] = m` Now, Alice can operate with the digest value of **[M]——> (m)** in cleartext because, as we have learned before, it is almost impossible to return to **[M]** from its cryptographic hash **(m)**.Alice calculates the signature **(S)** such that:

- **Step 1** – Making the inverse of **[k]** in **(mod p-1)**:

```
[INVk] ≡ k^(-1) (mod p-1)
```

- **Step 2** – Performing the equation:

`S ≡ [INVk]* (m - [a] *y1) (mod p-1)` Alice sends to Bob the public parameters **(m, y1, S)**.

- **Step 3** – In the first verification, Bob performs **V1**:

`V1 ≡ A^(y1) * y1^(S) (mod p)` After the decryption step, if **h[M] = m**, Bob obtains the second parameter of verification, **V2**: `V2 ≡ g^m (mod p)` Finally, if **V1 = V2**, Bob accepts the message.Now, let's better understand this algorithm with the help of a numerical example.**Numerical example**:Let's suppose that the value of the secret message is: `[M]= 88` We assign the following values to the other public parameters: `g = 7p = 200003h[M] = 77` The first step is the "Key initialization" of the private and public keys: `[b] = 2367 (private key of Bob)[a] = 5433 (private key of Al` the initialization process, Alice calculates the inverse of the key (*Step 1*) and then the signature (*Step 2*):

- **Step 1** – Alice computes the inverse of **[k]** in **(mod p-1)**:

```
[INVk] ≡ [k]^(-1) (mod p-1) = 23^-1 (mod 200003 - 1) = 34783
```

- **Step 2** – Alice can get now the signature **(S)**:

```
S ≡ [INVk]* (m - [a] *y1) (mod p-1)S ≡ 34783 * (77 – 5433 * 9091
```

sends to Bob the public parameters **(m, y1, S)= (77, 90914, 72577)**

- **Step 3** – With those parameters Bob can perform the first verification **(V1)**. Consequently, he computes **V2**. If **V2 = V1** Bob accepts the digital signature **(S)**:

```
V1 ≡ A^(y1) * y1^(S) (mod p)V1 ≡ 43725 ^ (90914) * 90914 ^ 72577
```

verification **(V2)**: `V2 ≡ g^m (mod p)V2 ≡ 7^ 77 (mod 200003) = 76561V2 = 76561` Bo verifies that **V1= V2**.Considering the underlying problem that makes this algorithm work, we can say that it is the same as the discrete logarithm. Indeed, let's analyze the verification function: `V1 ≡ A^(y1) * y1^(S) (mod p)` All the elements are made by discrete powers, and as we already know, it's a hard problem (for now) to get back from a discrete power even if the exponent or the base is known. It is not sufficient to say that *discrete powers and logarithms* ensure the security of this algorithm. As we saw in *Chapter 3, Asymmetric Cryptography*, the following function could also be an issue: `y2 ≡ [M]*B^[k] (mod p)` It's given by multiplication. If we are able to recover **[k]**, then we have discovered **[M]**.So, the algorithm suffers not only from the discrete logarithm problem but also from the factorization problem.Now that you have learned about the uses and implementations of digital signatures, let's move forward to explore another interesting cryptographic protocol: blind signatures.

Blind signatures

David Chaum invented **blind signatures**. He struggled a lot to find a cryptographic system to anonymize *digital payments*. In 1990, David funded *eCash*, a system that adopted an untraceable currency. Unfortunately, the project went bankrupt in 1998, but Chaum will be forever remembered as one of the pioneers of digital money and one of the fathers of modern cryptocurrency, along with Bitcoin.The underlying problems that Chaum wanted to solve were the following:

- To find an algorithm that was able to avoid the *double-spending problem* for electronic payments.
- To make the digital system *secure and anonymous* to guarantee the *privacy* of the user.

In 1982, Chaum wrote an article entitled *Blind Signatures for Untraceable Payments*. The following is an explanation of how the blind signatures described in the article work and how to implement them.Signing a message *blind* means to sign something without knowing the content. It could be used not only for digital payments but also if Bob, for example, wants to publicly register something that he created without making known to others the details of his invention. Another application of blind signatures is in electronic voting machines, where someone makes a choice (say, in an election for the president or for a party, for example). In this case, the result of the vote (the transmitted message) has to be known by the receiver obviously, but the identity of the voter has to remain a secret if the voter wants to be sure that their vote will be counted (that is the proper function of blind signatures).I will expose an innovative blind signature scheme for the *MBXI cipher* in *Chapter 6, New Algorithms in Public/Private Key Cryptography*, where I will introduce new ciphers in private/public keys, including the MBXI, invented and patented by me in 2011. Let's see now how David Chaum's protocol works by performing a blind signature with RSA.

Blind signature with RSA

Suppose Bob has an important secret he doesn't want to expose to the public until a determined date. For example, he discovered a formidable cure for coronavirus to avoid cancer and he aspires to get the Nobel Prize.Alice represents the commission for the Nobel Prize.Alice picks up two big secret primes **[pa, qa]**:

- **[pa*qa] = Na**, which is Alice's public key.
- **(e)** is the same public parameter already defined in RSA.

The parameter **[da]** is Alice's private key, given by this operation: `INV (e) ≡ [da] mod(pa-1)(qa-1)` Suppose that **[M1]** is the secret belonging to Bob. I have just called it **[M1]** to distinguish it from the regular **[M]**.Bob picks up a random number **[k]** and keeps it secret.Now Bob can go ahead with the blind signature protocol on **[M1]**:

- **Step 1** – Bob performs encryption **(t)** on **[M1]** to *blind* the message:

`t ≡ [M1] * [k]^e (mod Na)` Bob sends **(t)** to Alice.

- **Step 2** – Alice can perform the blind signature given by the following operation:

`S ≡ t^[da] (mod Na)` Alice sends **(S)** to Bob, who can verify whether the blind signature corresponds to the message **[M1]**.

- **Step 3** – Verification:

Bob calculates **(V)**: `S/k ≡ V (mod Na)` Then he can verify the following: `V^e ≡ [M1] (mod Na)` If the last operation is **TRUE**, it means Alice has effectively signed *blind* **[M1]**. In this case, Bob can be sure of the following: `[M1]^[da] ≡ V (mod Na)` Since no one except Alice could have performed function **(S)** without being able to solve the *discrete logarithm* problem (as already seen in *Chapter 3, Asymmetric Algorithms*), the signer must be Alice, for sure. This sentence remains valid until any other variable occurs; for example, when someone finds a logical way to solve the discrete logarithm or a quantum computer reaches enough qubits to break the algorithm, as we'll see in *Chapter 8*, *Quantum Cryptography*. Let's see an example to better understand the protocol.**Numerical example**:

- The parameters defined by Alice are as follows:

`pa = 67qa = 101` So, the public key **(Na)** is the following: `67*101 = Na = 6767da ≡ 1/e (mod (pa-1) * (qa-1))Reduce [`

- Bob picks up a random number, **[k]**:

`k = 29` Bob calculates **(t)**: `t ≡ M1* k^e ≡ 88 * 29^17 = 3524 (mod 6767)`

- Bob sends **(t)** to Alice ——————————————> Alice can now blind-sign **(t)**:

`S ≡ t^[da] ≡ 3524^1553 = 1533 (mod 6767)`

- Bob can verify **(S)** <—————————————— Alice sends **(S = 1553)**s

`S/k ≡ V (mod Na)1533/29 = 2853 (mod 6767)Reduce [k*x == S, x, Moc`
Bob accepts the signature **(S)**.As you can see from this example and the explanation of blind signatures, Alice is sure that Bob's discovery (the coronavirus cancer cure) belongs to him, and Bob can preserve his invention without declaring the exact content of it before a certain date.

Notes on the blind signature protocol

You can do a double-check on **[M1]**, so you will be able to realize that Alice has really signed **[M1]** without knowing anything about its value: `M1^[da] ≡ V (mod Na)88^1553 ≡ 2853 (mod 6767)` A warning about blind signatures is necessary, as Alice doesn't know what she is going to sign because **[M1]** is hidden inside **(t)**. So, Bob could also attempt to convince Alice to sign a $1 million check. There is a lot of danger in adospting such protocols. Another consideration concerns possible attacks.As you see here, we are faced with a factorization problem: `t ≡ M1* [k]^e (mod Na)` We can see that **(t)** is the product of **[X*Y]**: `X = M1 <——— Factorization problemY = k^e` It doesn't matter if the attacker is unable to determine **[k]**, as they can always attempt to find

[M1] by factoring **(t)**, if **[M1]** is a small number, for example. It's simply the case of **M1=0**, because of course **(t)** will be zero and the message can be discovered by the attacker to be **M = 0**.sOn the other hand, if we assume, for example, that **(k^e)** is a small number, since **[k]** is random, then the attacker can perform this operation: `Reduce [(k^e)*x == t, x, Modulus -> Na]x = MESSAGE` In this case, if, unfortunately, **[k^e] (mod Na)** results in a small number, the attacker can recover the message **[M1]**.As you will understand after reading the next chapter, blind signatures are the precursor to zero-knowledge protocols; the object of study in the next *Chapter 5, Introduction to Zero-Knowledge Protocols*. Indeed, some of the elements we find here, such as random **[k]** and the execution of blind signatures, and the last step of verification, **V = S/k**, performed by the receiver, utilize the logic that inspired zero-knowledge protocols.

Summary

In this chapter, we have analyzed hash functions, digital signatures, and blind signatures. After introducing hash functions, we started by describing the mathematical operations behind these one-way functions followed by an explanation of SHA-1. We then explained digital signatures with RSA and ElGamal with practical numerical examples and examined the possible vulnerabilities. Finally, the blind signature protocol was introduced as a cryptographic instrument for implementing electronic voting and digital payment systems.Therefore, you have now learned what a hash function is and how to implement it. You also know what digital signatures are, and in particular, you got familiar with the signature schemes in RSA and ELGamal. We also learned about the vulnerabilities that could lead to digital signatures being exposed, and how to repair them.Finally, you have learned what blind signatures are useful for and their fields of application. These topics are essential because we will use them abundantly in the following chapters of this book. They will be particularly useful in understanding the zero-knowledge protocols explained in *Chapter 5*, *Introduction to Zero-Knowledge Protocols*, and the other algorithms discussed in *Chapter 6*, *New Algorithms in Public/Private Key Cryptography*. Finally, *Chapter 6* will examine new methods of attack against digital signatures. Now that you have learned the fundamentals of Hash functions and digital signatures, it is time to analyze zero-knowledge protocols in detail in the next chapter.

5 Introduction to The “spooky math”. Zero-Knowledge Protocols. and Attacks

As we have already seen with the digital signature, the authentication problem is one of the most important, complicated, and intriguing challenges that cryptography is going to face in the near future. Imagine that you want to identify yourself to someone who doesn't know you online. First, you will be asked to provide your name, surname, and address; going deeper, you will be asked for your social security number and other sensitive data that identifies you. Of course, you know that exposing such data via the internet can be very dangerous because someone might steal your private information and use it for nefarious purposes.Some time ago, I watched a video that impressed me. I have even decided to insert it into my presentations about privacy and security, and I presented it during an event related to Smart Cities in Silicon Valley where I had been invited to talk. The video starts with an alleged magician who invites people to enter a tent set up in the middle of the city. The magician reads, one by one, each person's past and something about their future. The unbelievable thing was that the magician (who had never met any of the people interviewed before) knew particulars about the astonished participants' lives only they were supposed to know.How was it possible? Was it really magic or was it just a trick? At the end of the video, the trick was revealed. The magician knew everything about the participants' lives – finances, assets owned, and even credit card numbers – thanks to a staff of techno-hackers who sat behind a curtain, working hard to discover all the digital secrets they could about the participants. If a hacker knows your identity, they can easily find out about most of your digital life.In another case, you might have read a news story where a gang of thieves planted a fake ATM in a commercial center. Each time a person inserted a card and entered their PIN, a computer recorded this information and the ATM refused the operation. Once the information had been collected from the cards and the PINs stolen, the hackers could then clone the cards, reproducing them along with their PINs, and subsequently be able to withdraw money at an actual ATM.How is it possible to block this kind of scam? There are many situations in which sensitive information, such as passwords and other private information, is required. If a hacker obtains this information linked to a person or a machine, they can easily steal identities and wreak havoc for their victims.One way to solve these kinds of problems is by not revealing any sensitive information, such as: Name, Surname, address, social security number, PIN , but this is not always possible. Another way is to avoid exposing private information by giving a *proof of knowledge*. A proof of knowledge is essentially a way to bypass to give more information than necessary or identify yourself not exposing your personal data. These cryptographic protocols are called **Zero-Knowledge Protocols** (**ZKPs**), which we are going to study in this chapter.In this chapter, we are going to cover the following topics:

- Mainframe and logic of Zero-Knowledge Protocols
- Non-interactive and interactive ZKPs (the Schnorr protocol) with examples and possible attacks on them
- SNARK protocols and Zcash in a nutshell
- One-round ZKPs
- A new Protocol by the Author: ZK13 and the zero-authentication protocol

Now that you have been introduced to the world of ZKPs and know what they are used for, it's time to go deeper to analyze the main scenarios and protocols used in cryptography.

The main scenario of a ZKP – the digital cave

Imagine this fantastic scenario: Peggy has to demonstrate to Victor that she is able to open a locked door in the middle of *Ali Baba's cave*, an annular cave with only one entrance/exit that can be reached from two directions, as you can see in the following figure. I suppose you have noticed that I have changed the names of the two actors, Peggy and Victor, from the usual Alice and Bob, just because here a verification is due, and the names **Peggy** and **Victor** match better with the first letters of **prover** (**P**) and **verifier** (**V**).

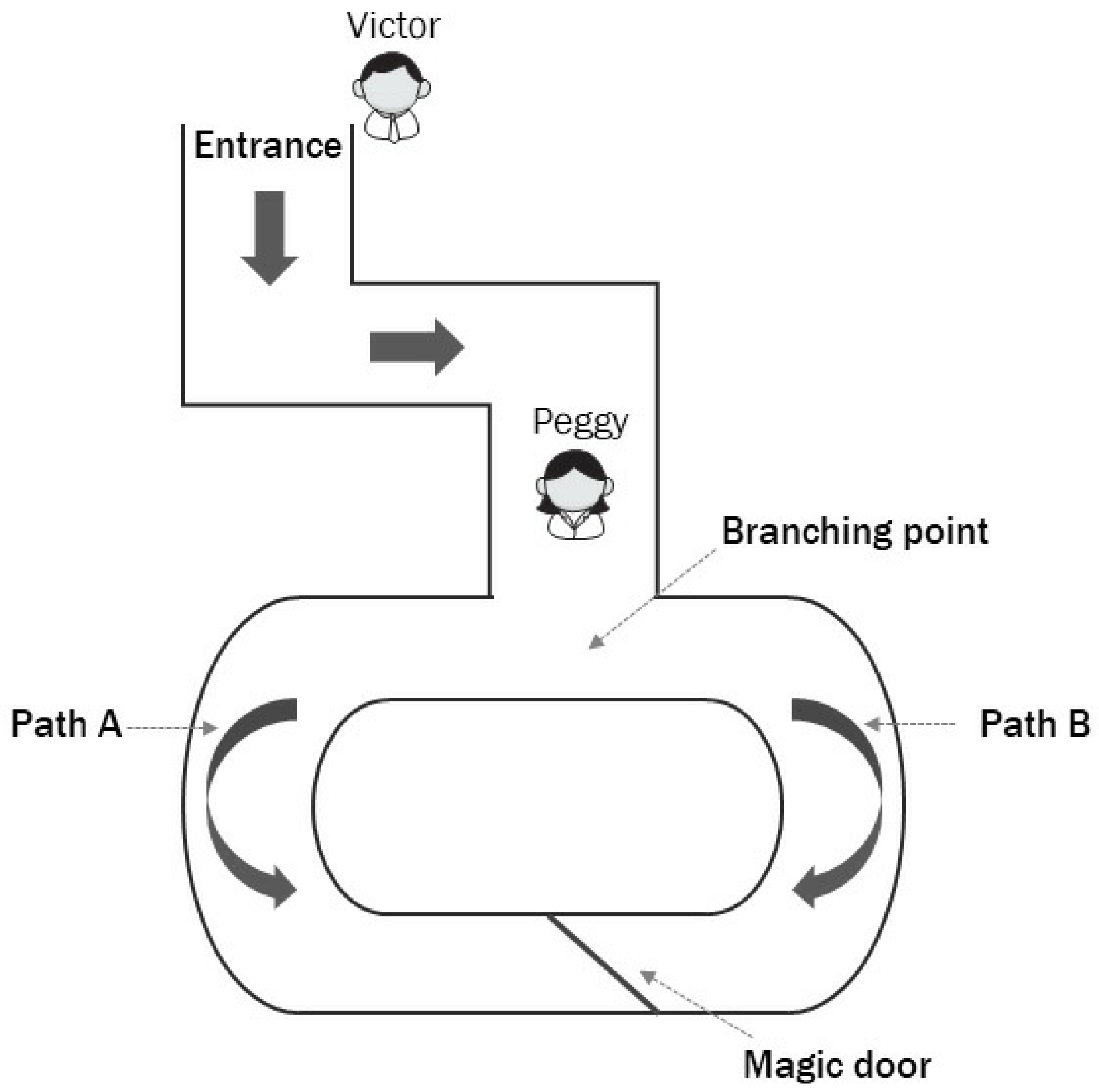


Figure 5.1 – Ali Baba's cave

This example involves Ali Baba's cave revisited in modern times, in which the door in the middle of the cave is locked through a secret electronic combination that's strong enough to prevent the entry of anyone who doesn't know the secret combination. Now, suppose that Peggy must prove to Victor that she knows the combination to unlock the door without revealing the numbers to him.So then, the challenge for Peggy is *not* to reveal the combination of the door to Victor, because Peggy is not sure whether Victor knows it and she doesn't want to give Victor any information about the combination. She just needs to demonstrate to Victor that she can exit from the opposite side of the cave.Expressed differently, ZKP is a challenge where the answer is not revealing the exact information required, but simply proving to be able to solve the underlying problem. Indeed, the natural way to demonstrate knowing something is to reveal it, and the verification just comes naturally. However, by taking this approach of just demonstrating what you know so directly, you could reveal more information than required. With a ZKP, Peggy can avoid the issue of revealing the digital door's combination and, at the same time, Victor can be sure (even if he doesn't know the combination) that Peggy knows the combination if he sees Peggy coming out of the cave from the opposite side. There are many ways to implement a ZKP because, as you can imagine, there are many scenarios in which such verification could be required. For example, you can think of Peggy as a human and Victor as a machine (an ATM or server). Peggy has to identify herself to the machine, but she doesn't want to reveal any sensitive data, such as her name or surname. She just has to prove to the machine that she is really who she is supposed to be. The aim, in this case, is to avoid revealing Peggy's identity. ZKPs can be applied here. Another use case where ZKPs can be applied is the authentication of virtual machines in a computer network. We will cover this use case in *Chapter 9, Crypto Search Engine*, where we will use ZKPs to protect against man-in-the-middle attacks.ZKPs can be applied to other kinds of challenges than authentication, such as the fields of nuclear disarmament and blockchain. Now, we will delve deeper into the applications of ZKPs, starting with analyzing non-interactive protocols used to prove statements.

Non-interactive ZKPs

The protocol we are going to analyze in this section is a **non-interactive ZKP**. This means that the prover has to demonstrate the statement, assuming

that the verifier does not know the solution (the content of the statement) and that the verification is made without any exchange of information from the verifier.The scheme could be summed up as follows:**Prover** (statement) —————-> [**Proof of knowledge**] ——————> **Verifier** (verification)Let's take this problem into consideration: Peggy states to know that a document **[m]** is encrypted with RSA, as follows: `m^e ≡ c (mod N)But Peggy doesn’t want to reveal the conten`
(**N**, **c**, **e**) are public parameters, and **[m]** is secret.

Important Note

> Remember, I always denote the secret elements of the functions with the **[]** symbols.

In order to demonstrate the statement to Victor, the following protocol is executed:

- Peggy chooses a random integer, **[r1]** (she keeps it secret), and calculates the inverse of **r1** (represented as **INV[r1]**) multiplied by **[m]** (modulo **N**):

```
r2 ≡ [m] * r1^-1 (mod N) [Peggy keeps r2 secret too]
```

- Peggy calculates (**x1**) and (**x2**), as follows:

`x1 ≡ r1^e (mod N)x2 ≡ r2^e (mod N)` Peggy sends **x1** and **x2** to Victor.

- Finally, Victor verifies the following:

`x1*x2 ≡ c (mod N)` It is supposed here that if Victor can verify *step 3*, **x1*x2 ≡ c (mod N)**, then Peggy really should know **[m]**.As you can imagine see, Peggy wants to demonstrate to Victor that she effectively knows the message **[m]** without revealing it. Remember that in this case, Peggy ignores whether Victor knows **[m]** or not. In fact, using this protocol it’s irrelevant if Vicor actually knows [**m**] or not. So, the underlying challenge implicit to this statement is for Peggy to be able to solve the RSA problem without revealing **[m]**. Indeed, it's supposed that if Peggy knows **[m]** (hidden in the cryptogram (**c**)), she can also calculate the function **x1*x2 ≡ c (mod N)**; otherwise, she will not be able to do that.Another important consideration is the following: if (**c**) is a big number, it is supposed (I hope you will agree with me, but don't take this as gospel) that it should be hard to know **x1** and **x2** (the two factors of (**c**)) without knowing **[m]**. It could be considered the same degree of computational difficulty to factoring (**N**). As you can see in the preceding function, (**r2**) is calculated using **[m]** in the following equation: `r2 ≡ [m] * r1^-1 (mod N)` So, the final effect of the multiplication between **x1*x2** (as you can see in the following demonstration) will be to eliminate (**r1**) and leave **m^e (mod N)**, which is equal to (**c**).Even if Victor doesn't know **[m]**, he can believe what Peggy states (to know **[m]**) because she has demonstrated she can *factorize* (**c**).RSA is supported by the factorization problem (as we saw in *Chapter 3, Asymmetric Encryption*); here, the function is as follows: `x1*x2 ≡ c (mod N)` This states that it is computationally hard to find two numbers (**x1**, **x2**) that factorize (**c**).

Important Note

> I will prove that an attack on this protocol exists that avoids the factorization problem in order to trick Victor, which we will experiment with later in this chapter.

Numerical example:Let's see with a numerical example how this protocol works before going deeper to analyze it: `m = 88N = 2430101e = 9007m^e ≡ c (mod N)88^9007 ≡ 160613 (mod 2` let's start the protocol of verification.Peggy chooses a random number: `r1 = 67`

- **Step 1**: Peggy calculates **r2**:

`r2 ≡ [m] * r1^-1 (mod N)` First, Peggy calculates the **[Inv(r1)]** function (the inverse value of **r1 (mod N)**), and then she multiplicates it for **[m]**: `67 * x ≡ 1 (mod 2430101)x = 217621 [m]* x ≡ r2 (mod N)88 * 21` we have Peggy gets **r2**: `r2 = 2139941`

- **Step 2**: Then, Peggy calculates **x1** and **x2**:

```
x1 ≡ r1^e (mod N)x2 ≡ r2^e (mod N)67^9007 = 1587671 (mod 2430101)
```

Peggy sends **x1**, **x2** (**1587671**, **374578**) to Victor.

- **Step 2**: Finally, Victor can verify the following:

`x1 * x2 ≡ c (mod N)1587671 * 374578 ≡ 160613 (mod 2430101)160613`
see why this protocol is mathematically correct.

Demonstration of a non-interactive ZKP

We have to demonstrate the following: `x1 * x2 ≡ c (mod N)` Where **c** ≡ **[m]^e (mod N)**, we can substitute in the previous function (**x1 = r1^e**) and (**x2 = r2^e**) so that we get the following: `x1 * x2 ≡ r1^e * r2^e ≡ c (mod N)` Substituting **r2** into the equation, we have the following: `x1 * x2 ≡ (r1)^e *(m * ((r1^-1)^e) ≡ (r1)^e * m^e * (Inv` by the modular power's properties (collecting together the two factors I have highlighted), we have the following: `r1^e * (Inv(r1))^e ≡ 1 (mod N)` So, eliminating **r1^e** and **(Inv(r1))^e** from the final equation will leave only the **m^e** remaining in the second stage of the equation, and the result will be as follows: `x1 * x2 ≡ m^e ≡ c (mod N)` As you know, since the beginning of this demonstration, (**m^e**) is just the RSA encryption of the secret message **[m]**, which is equal to the cryptogram (**c**); that is why **x1*x2 = c**. That's just what we wanted to demonstrate.This protocol has an important characteristic: it's executed in only 2 Steps avoiding any interaction between Peggy and Victor:Peggy calculates the parametersVictor verify the correctness The next section will show how we can attack an RSA ZKP.

Demonstrating an attack on an RSA ZKP

If you have stayed with me until this point, I'm hoping you will follow me further on this journey, so I can give you a demonstration of using a protocol to trick the verifier.Note that I created this attack at the end of 2018. This is one of the possible attacks on a ZKP that I have demonstrated.The goal of this attack is to demonstrate that Eve (the attacker) can calculate two *fake* numbers (**x1**, **x2**) which prove to factorize (**N**) even if Eve effectively doesn't know **[m]**.Let's explore how this attack works and what effects are produced:

- Eve (the attacker) picks up a random number, **[r]**, and calculates the following:

`[r] * (v1) ≡ e (mod N)` **[r]** and (**e**) are public, so that is known by everyone. By means of this function, Eve can extract (**v1**).

- In parallel, Eve calculates the following:

`e * x ≡ c (mod N)` The parameters (**e**, **c**) are also known. This, just like in *step 1*, is an inverse multiplication (modulo **N**). The scope of this operation is to obtain **[x]**. Then, using **[x]**, Eve multiplies **[x]** by **[r]**, yielding (**v2**): `x * r ≡ v2 (mod N)` Eve sends (**v1**, **v2**) to Victor, who can verify the following: `v1 * v2 ≡ c (mod N)` Eve can impersonate Peggy, and she can claim to know **[m]** even if she doesn't know it!**Numerical example**:**r = 39** is the secret number chosen by Eve. (**N**, **c**, **e**) are the same public parameters of the previous example (**N = 2430101**, **c = 160613**, **e = 9007**).

- **Step 1**: Eve calculates **v1**:

```
e * r^-1 ≡ v1 (mod N)9007 * 436172 = 1557988 (mod 2430101)v1 = 15
```

- **Step 2**: Eve performs **v2**.

The next operation is to obtain the inverse of (**e**) with respect to (**c**).**e * x ≡ c (mod N)**, obtaining (**x**) in inverse modular multiplication: `9007 * x = 160613 (mod 2430101)x = 2031892` Then, using **x**, Eve obtains gets **v2**, performing the following operation:**x * r ≡ v2 (mod N)**, obtaining **v2**: `v2 = 1480556` After having gained **v2**, Eve sends (**v1** = **1557988**; **v2** = **1480556**) to Victor.

- **Step 3 2**: Verification stage.

Finally, Victor can verify the following: `v1 * v2 = c 1557988 * 1480556 = 160613 (mod 2430101)` The attack was successful!

Important Note

Peggy herself could be the primary actor of this trick if she doesn't know **[m]**, but she wishes to convince Victor of it.

This attack works because (**c**) contains **[m]**, and I don't need to demonstrate showing the value of **[m]**. This protocol isn't required to show **[m]** or its hash, **[H(m)]**, because Peggy doesn't want to reveal any information about **[m]** to Victor. Remember this is *not* an authentication protocol, but it's a proof of statement (or knowledge) that Peggy knows **[m]**.For completeness, there is to add that (c) is not a product of 2 big prime numbers (such as N is) as probably you have already noticed in the example. So, the logical base of this protocol is weak: finding 2 numbers whose product is (c) could be not so difficult even if (c) is a large number. To use a hypothetical example, you can think of a scenario where there are two countries: (A) has to demonstrate to (B) that it holds the formula for an atomic bomb. Using this ZKP, (A) could claim to know **[m]** (the formula of the atomic bomb) without really knowing it. This attack could be avoided under one certain conditions, one of those is the following:If Victor already knows **[m]**, then he can require Peggy to send him a hash of the message, **H[m]**. Victor can then verify whether (**x1** and **x2**) are the correct values, and he will accept or deny the verification based on the correspondence of the hash value with **[m]**.In this case, the problem is that the aim of this protocol was not to prove something that was already known but to prove something independently, regardless of whether or not it was known.This last point is very important because if Victor knows **[m]**, then this protocol works; if Victor doesn't know **[m]**, this protocol fails.To avoid this those problems, we have to switch to an *interactive* protocol, as we will see in the next section.

Schnorr's interactive ZKP

The protocol that we saw in the previous section is a non-interactive protocol, where Peggy and Victor don't interact with each other but there is simply a *commitment* between them. The commitment is that Peggy has to shows Victor that she knew the message **[m]** without revealing anything about it. Thus, she tries to demonstrate to Victor that she can overcome the RSA problem (or another hard mathematical problem) as proof of her honesty. However, we have also seen that this protocol can be *bypassed* using a mathematical trick.Let's see whether the following interactive ZKP is more robust and can prevent possibly devastating attacks.We always have Peggy and Victor as our main actors. So, let's assume the following:

- **p** is a big prime number.
- **g** is the generator of (**Zp**).
- **B** ≡ **g^a (mod p)** is the public parameter of Peggy.
- (**p**, **g**, **B**) are public parameters.
- **[a]** is the secret number object of the commitment.

Peggy claims that she knows **[a]; let's say that [a] it's the password to unlock a certain amount of money in a wallet.** In order to demonstrate the claim, Peggy and Victor apply the following interactive protocol:

- **Step 1**: Peggy chooses a random integer, **[k]**, where **1 ≤ k < p-1**.

She performs the following calculation: `V ≡ g^k (mod p)` Peggy sends (**V**) to Victor.

- **Step 2**: Victor chooses a random integer, (**r**), where **1 ≤ r < p-1**.

Victor sends (**r**) to Peggy.

- **Step 3**: Peggy calculates as follows:

`w ≡ (k - a*r) (mod p-1)` Peggy sends (**w**) to Victor.

- **Step 4**: Finally, Victor verifies as follows:

`V ≡ g^w * B^r (mod p)` If that is true, Victor should be convinced that Peggy knows **[a]**.Let's see why the protocol should work and the reason why the last function (**V**) states that Peggy really knows **[a]** (the commitment).First of all, I will show why the protocol is mathematically true, and then I will give a numerical example of this protocol.

A demonstration of an interactive ZKP

Recall the following instructions: `V ≡ g^k (mod p) B ≡ g^a (mod p)w ≡ k - a*r (mod p-1)` N
we substitute all the past equations inside the last verification of *step*

3: `V ≡ g^w * B^r (mod p)V ≡ g^k (mod p)` Substituting the functions in (**V**), the equation becomes the following: `V ≡ (g^ (k - a*r (mod p-1))) * ((g^a)^r) (mod p)` For the properties of exponential factors, we have the following: `g^k ≡ g^ (k -[ar]) * g^[ar] (mod p)V ≡ g^k ≡ g ^(k -ar +` **[-ar]** with **[+ar]**, we get back the following: `g^k ≡ g^k (mod p)` That's what we wanted to demonstrate.Let's do a numerical example to better visualize how this interactive ZKP works. **Numerical example**: `p = 1987a = 17g = 3` (**p = 1987** and **g = 3**) are public parameters. **[a]** = 17 is the secret number that Peggy claims to know: `B ≡ g^a (mod p)` (**B**) is the public key of Peggy, given by the following: `3^17 ≡ 1059 (mod 1987)` Peggy picks up a random number, **[k]** = 67, and she calculates (**V**): `V = 3^67 = 1753 (mod 1987)` Peggy sends (**V**) to Victor.Victor picks up a random number (**r** = 37) and sends it to Peggy, who can calculate was following: `k – a * r ≡ w (mod p-1)67 – 17 * 37 ≡ 1424 (mod 1987-1)w` sends (**w** = 1424).Finally, Victor now verifies whether (**V**) = 1753 corresponds to the following: `g^w * B^r ≡ V (mod p)3^1424 * 1059^37 ≡ 1753 (mod 1984)V` fact, it does correspond.Now, we analyze the reason why this protocol states that by knowing **[a]** automatically, Peggy can convince Victor. Let's use an example to better understand the problem.This protocol can be used as an *authentication scheme* in which, for example, Victor is a bank that holds the public parameter **(B)** of Peggy (a client of the bank). The secret number **[a]** could be Peggy's secret code (PIN). In order to gain access to her online account, Peggy has to demonstrate that she knows **[a]**. In another use case, we could have Victor as a central unit computational power (server) and Peggy as a user who wants to connect to the server using an insecure line, or again (as we will see later), Peggy could be another server, too.The point of using a ZKP is to avoid Peggy revealing her sensitive data to the public. So, the underlying problem she has to demonstrate to Victor consists of solving a challenge in which she can demonstrate that she knows the discrete logarithm of **(B)**. As we have seen in *Chapter 3*, *Asymmetric Encryption*, knowing **(B)** and **(g)** is not enough to compute **[a]** in this function: `B ≡ g^[a] (mod p)` This is because we are operating in modular functions.Of course, Peggy needs to know **[a]** if she wants to compute the verification function, **(w)**: `w ≡ k - a*r (mod p-1)` There is no way to trick

Victor, who moreover sent (**r**) to Peggy, which is used in the last verification function together with (**v**) and (**B**), along with (**r**), to be sure that Peggy cannot bluff: `V ≡ g^w * B^r (mod p)` So, I hope to have convinced you that there is no way for Peggy to trick Victor in this case.

Demonstrating an A Challenge for a disruptive attack on an interactive ZKP

Now that we have seen that Peggy can't trick Victor, I have propose an attack against this protocol that I created in late 2018; let's see whether it works or not.Here, the scope for an attacker (Eve) is to provide a final proof of verification without knowing the secret number, **[a]**.**Step 1**: Peggy chooses a random integer **[k]** where **1 ≤ k < p-1**.Then she calculates the following: `V ≡ g^[k] (mod p)` It's when Peggy sends (**V**) to Victor that Eve can try to shoot a man-in-the-middle attack.Peggy sends (**V**) to Victor.**Step 2**: Victor chooses a random integer, (**r**), where **1 ≤ r < p-1**.Eve injects **V1 ≡ g^k1 (mod p)**, where **[k1]** is a number invented by Eve that substitutes **[k]**.Eve sends (**V1**) to Victor, substituting her value for Peggy's result. After receiving (**r**) from Victor, Eve calculates (**v1**) as follows: `V1 ≡ g^(v1) * B^r (mod p)` This is the path to the attack:

- If you can exclude (**r**) from the final verification function, (**V1**), then you have reached the goal.
- Essentially, the attacker should find a value for (**v1**), as follows:

`v1 = [x] ———-> V1 = g^k1` Here are some notes:

- Remember that you don't have to implement (**v1**) in the same way the preceding function (**v**) did, but you are free to give (**v1**) any value.
- Remember that the earliest point of attack is substituting (**V**) with (**V1**), but that is not mandatory. In this case, the warning is that as you don't know (**r**) when you have delivered (**V1**) to Victor, this parameter can no longer be changed.
- Good luck! If you are able to find a way to trick the Schnorr interactive protocol, please let me know when you have arrived at a conclusion, and you will get a cryptographer researcher position.

The preceding analyzed interactive protocol suffers from another problem. Let's imagine that two people live in different time zones, such as Europe and Australia. If one is *ON*, the other one is probably *OFF* because they're he/her is sleeping. So, this isn’t probably the most appropriate protocol to use. what happens if they have to wait for many hours to make or receive an economic transaction?This protocol doesn't fit well with this kind of purpose, such as cryptocurrency transactions. Most cryptocurrency protocols use zero-knowledge algorithms to anonymize data inside their architecture structures. Now that we know how to implement such a protocol, we can explore zk-SNARKs.

An introduction to zk-SNARKs – spooky moon math

If you think ZKPs are pretty difficult to understand, that is because you haven't yet faced off with **zk-SNARKs** – are a kind of non-interactive ZKP, that is a little bit complicated so that are also known as **spooky moon math**. Here, the situation gets a little bit more complicated, but don't worry – it's not impossible. In the next section, you will see interesting new attack possibilities.**Non-interactive zero-knowledge proofs**, also known as **zk-SNARKs** or **zk-STARKs**, are kinds of ZKPs that require no interaction between the prover and verifier, like the first protocol we saw in this chapter. In this section, we are going to focus on zk-SNARKs.The name zk-SNARK stands for **Zero-Knowledge Succinct Non-Interactive Argument of Knowledge**. So, we are facing off with schemes that need only one interaction between the prover and the verifier.Indeed, zk-SNARKs are very much appreciated for their ability to anonymize transactions and to identify users in cryptocurrency schemes, as we will see in this section.The first cryptocurrency that zk-SNARKs have been adopted in the blockchain as a scope of authentication and this new system to create *consensus* .was **Zcash**. The use of zk-SNARKs in a blockchain is important, as we will see later, for the use of smart contracts. As you may know, a smart contract is an escrow of cryptocurrency activated following the completion of an agreed execution.Since smart contracts and blockchains are not a part of this book, I will show just a limited example of how zk-SNARKs work in a cryptocurrency environment, as it will be useful to understand.For example, suppose Peggy makes a payment in Ethereum to execute a smart contract with Victor. In that case, both Peggy and Victor want to be sure that the execution of the smart contract (for Peggy) and the payment received (for Victor) are completed successfully. However, many details inherent to the smart contract will not be revealed. So, the role that zk-SNARKs play is fundamental to covering these secrets and executing smart contracts. In order to work, the protocol has to be fast, secure, and easy to implement.As we have already seen, you will notice that this is just what the purpose of a ZKP is – to ease the navigation of an untrustworthy environment. Here, we are talking about blockchains and virtual payments, but essentially the process is similar.So, in this environment, zk-SNARKs keep secrets by protecting the steps involved in a smart contract and, at the same time, proving that all these steps have been executed. This way, they protect the privacy of people and companies.Remember that – not because you have to be super-skeptical, but because you should be realistic – this statement is true under determinate conditions, which I will try to explain as follows:

- The proof given by the prover holds the same computational degree of difficulty as the underlying algorithm chosen as a proof of knowledge.
- There is no mathematical way to trick the verifier with a shortcut or fake proof (such as substituting fake parameters into the **V1 ≡ g^(v1) * B^r (mod p)** equation in order to avoid knowing **[a]**).

So, let's see how a zk-SNARK works.

Understanding how a zk-SNARK works

In this section, first of all, I will try to synthesize how zk-SNARKs generally work, and then we will return with a zk-SNARK protocol related to a proof of knowledge based on a discrete logarithm.As we already have seen for the other ZKPs, a zk-SNARK is composed of three parts or items – (**G**), (**P**), and (**V**):

- **G**: This is a generator of keys, made by a private parameter (the statement or another random key) that generates public parameters given by private keys.
- **P**: This is a proof algorithm that states what the prover wants to demonstrate.
- **V**: This is a verification algorithm that returns a **TRUE** or **FALSE**

Boolean variable from the verifier. I will demonstrate now that using ZKPs (and, in particular, zk-SNARK protocols) is *not* enough to keep **[w]** secret, but it is possible to arrive at proving the statement as **TRUE** if it is also **FALSE**.

Let's look at how a similar protocol example explained in the *Interactive ZKP* section (Schnorr) would work in a non-interactive way (zk-SNARK mode).In this protocol, we have Anna as the prover and Carl as the verifier.Here, Anna has to prove that **[a]** is known to her.Anna calculates her private key, (**y**), given by the following: `y ≡ g^a (mod p)` (**g**) is a generator (as in D-H or other private-public algorithms we have already seen in this book).Then, Anna picks up a random value, **[v]**, inside **p-1**, which she keeps secret, and consequently, she can calculate the following: `t ≡ g^v (mod p)` Anna calculates (**c**) as a hash function of the three parameters, (**g**, **y**, **t**), and she can compute (**r**) as follows: `r ≡ v – c*a (mod p-1)` The verifier, Carl, can check the following: `t ≡ g^r * y^c (mod p)` Finally, if the verification validates the two terms of the function, then Carl accepts that the statement **[a]** proposed by Anna is **TRUE**.Now that we have analyzed how this ZKP works in a zk-SNARK environment, let's see an attack on this protocol before we cover a numerical example.

Demonstrating an attack on a zk-SNARK protocol

This attack was performed by me in June 2019 and just goes to show that nothing is completely secure.Let's say that Eve is an **Artificial Intelligence** (**AI**) server. We suppose that Eve (AI) intercepts the **H(g, y, t)** public hash function and performs a MiM attack. While Peggy sends **(V)** to Victor, Eve substitutes **(c) = H(g, y, t)** with **(c1) = H1(g, y, t1)**, remembering that (**H**) is the hash function and that (**t1**) is given by the following: `t1 ≡ g^v1 (mod p)` As you have probably noticed, substituting (**v**) with (**v1**) is the same trick that substituted (**k**) with (**k1**) in the previous attack.Simply, Eve can put (**r1**) as follows: `r1 = v1` Now, Eve orders the AI (the intelligent third server connected to the internet) to send to Carl (**r1**, **v1**, **c1**), who can verify the following: `t1 ≡ g^r1 * y^c1 (mod p)` It's simple to demonstrate that **t1 = g^v1** because of the following: `y^(p-1) ≡ 1 (mod p)` Finally, as we have assigned **c1 = p-1** and **r1 = v1**, the final effect will be as

follows: `t1 ≡ g^v1 ≡ g^v1 * 1 (mod p)` **Numerical example**: `p = 3571g = 7x = 23` Anna's public key is as follows: `y ≡ g^x (mod p)7^ 23 = 907 (mod 3571)y = 907` Now, I will show you how Anna can demonstrate to Carl to get the secret number, **[x]**. She chooses **v = 67**, as follows:
`t ≡ g^v (mod p)7^ 67 = 584 (mod 3571)t = 584` Let's suppose that hash (**g**, **y**, **t**) as follows: `c = 37` She computes **r** as follows: `r ≡ v - c*x (mod p-1)(67 - 37*23) ≡ 2786 (mod 3570-1)r = 2`
sends **(r, t, c) = (2786, 584, 37)** to Carl. Carl can verify the following: `g^r * y^ c ≡ t (mod p-1)7^2786 * 907^37 = 584 (mod 3571)`
the AI intercepts the public parameters (**y**), (**t**), and (**r**). Eve leaves the (**y**) invariant, but she changes (**t**) with (**t1**) and (**r**) with (**r1**), performing a man-in-the-middle attack: `v1 = 57` Eve calculates the following: `t1 ≡ 7^57 (mod 3571)t1 = 712v1 = r1 = 57c1 = p-1 = 3570`
sends **(r1, t1, c1) = (57, 712, 3570)** to Carl. Carl verifies the following: `t1 ≡ g^r1 * y^c1 (mod p)` I highlighted the parameter that Eve substitutes, **(t1, r1, c1)**; she left the (**y**, **g**, **p**) invariant.Substituting the new parameters into the equation of verification, we have the following: `7^57 * 907^3570 ≡ 712 (mod 3571)` Indeed, Carl is able to verify that **t1 = 712** corresponds with the parameters received from Eve.Essentially, if Carl is not able to recognize that **r1 = v1** and/or he doesn't accept **c = p-1**, then the trick is done, and Eve can replace Anna.So, what are the protections to adopt against this attack? If this attack is implemented in a more sophisticated mode, it will probably be very difficult to avoid it.Note that the parameter (**y**), the public key of Anna that "envelopes" the private key **[a]** object of the statement, hasn't been modified during the attack.Anyway, zk-SNARKs can be implemented using other methods and protocols to prove statements; we will see what these algorithms and protocols are in the next section. Blockchains and cryptocurrency are evolving quickly to find new methods to authenticate users anonymously. However, with this topic being relatively new, it is better to make the effort to find all the possible attacks and the repair methods for them.

How to use Zk-SNARKs in Zcash cryptocurrency in a nutshell

In this section, we will analyze the zk-SNARK behind Zcash, a new in cryptocurrency giving you the basic information necessary to discover this world.that aims to preserve the security and privacy of transactions, as discussed in a scientific paper released on November 12, 2020 (*Demystifying the Role of zk-SNARKs in Zcash*):"The underlying principle of the Zcash algorithm is such that it delivers a full-fledged, ledger-based digital currency with strong privacy guarantees and the root of ensuring privacy lies fully on the construction of a proper zk-SNARK."A blog about Zcash stated the following regarding zero-knowledge proofs: *allow one party* (the prover) *to prove to another* (the verifier) *that a statement is true, without revealing any information beyond the validity of the statement itself*. In a version of zero-knowledge called "proof of knowledge," the prover can demonstrate a statement without revealing any sensitive information about the content. But, as we have seen until now, these protocols are based on problems that are difficult to solve, such as factorization and discrete logarithms, and we have seen in the previous sections how some of them could be vulnerable to various kinds of attacks. A reason why you should adopt a ZKP is to authenticate yourself without revealing too much information about your sensitive data or your server's identity. Another reason for using a ZKP is to prove that you know *something* that you want to keep secret. Here, we have two main different cases:

- In the first case, it is supposed that the verifier doesn't know the content of the statement.
- In the second case, it is supposed that the verifier already knows the content of the statement, and only has to verify its *correctness*.

All that becomes much more complicated if you decide to adopt a *non-interactive protocol* instead of an *interactive protocol* because, in the first case, the verifier does not need to give any further input. Indeed, as you can see, if a parameter is exchanged between the prover and the verifier, that could increase the security of the algorithm. On the other hand, we know that interactive protocols need many steps to reach the goal, hence they are not used as much as non-interactive protocols. As we have seen in the previous sections, I have shown some attacks that are capable of deceiving the verifier, making them believe something that isn't true; moreover, if the verifier doesn't know the content of the statement, it is even *easier* to send fake mathematical parameters to trick them.Now, the situation is changing fast in cryptography, for two reasons:

- The need for cryptography to address privacy and anonymize data for an increasing number of transactions over the internet and in e-commerce
- The rise in the use of digital payments and cryptocurrency

To anonymize exchanges of digital money and to ensure the privacy of users in cryptocurrency transactions, zk-SNARKs are adopted in several cases. The verifications used for the two main problems we have seen (RSA and the discrete logarithm) are almost obsolete. They are leaving room for new and more sophisticated, hard-to-solve problems; in Zcash, we see one such use case.To get perfect zero-knowledge privacy, Zcash implemented a complex system of functions to determine the **validity of a transaction**. In Zcash's consensus system, using zk-SNARKs, the objective is to return a response on the validity of a transaction without knowing anything about the content and terms of the transaction itself.Zcash adopts a complex scheme to anonymize transactions and obtain a final answer as to whether transactions are valid. Here, I have tried to schematize the circuit:Computation → Arithmetic circuit → R1CS → QAP → zk-SNARK

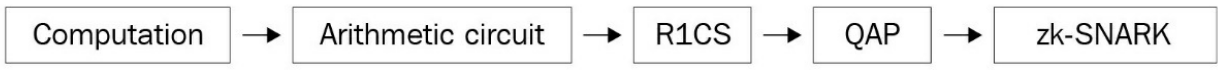


Figure 5.2 – Circuit flow

Let's analyze what these functions represent one by one.The first step is turning transaction validity into a mathematical function, which finally has to be gathered into a logical expression. This step is taken by creating an arithmetic circuit similar to a Boolean circuit (we saw this in *Chapter 2, Introduction to Symmetric Encryption*, and *Chapter 3, Asymmetric Encryption*) made by mathematical base operations (**+**, **-**, *****, and **/**) and computed by Boolean operators (**AND**, **OR**, **NOT**, and **XOR**), so that the program converges into only one *gate*, which is the result of all the operations chained together, as we can see in the following example of computing the expression **(a+b)*b*c**:

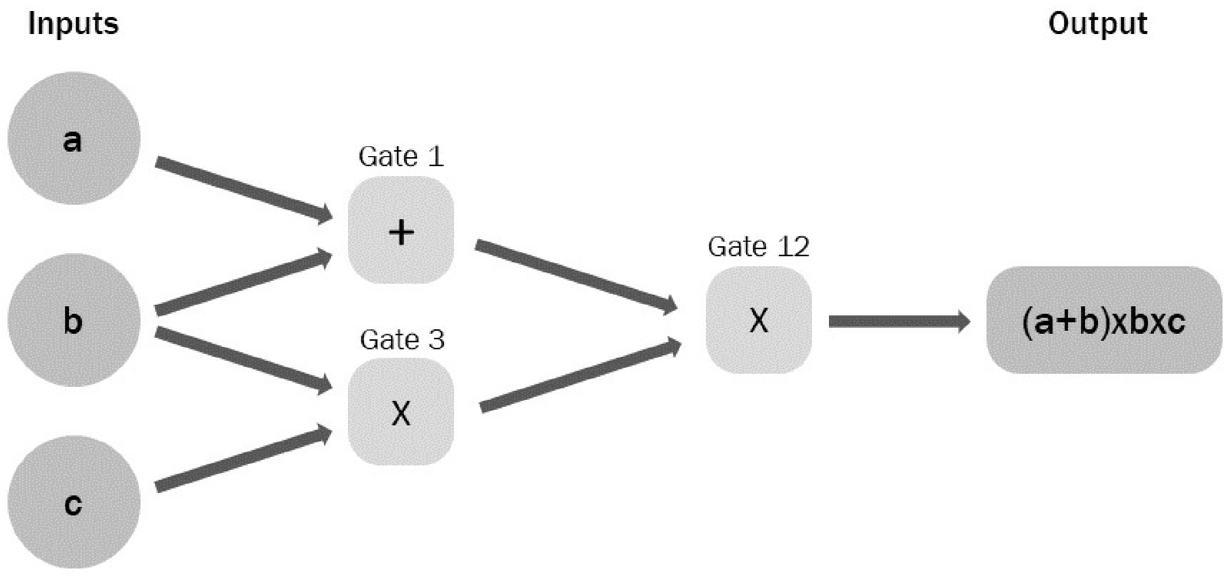


*Figure 5.3 – An example of an arithmetic circuit. Inputs: a, b, and c. Output: (a+b)*b*c*

As you can see, the **arithmetic** (or **algebraic**) circuit converges in only a single gate all the operations performed on the inputs: a, b, and c. Looking at this circuit, from left to right, we have the single terms (**a**, **b**, and **c**); first, **(a)** is added to **(b)**, then **(b)** is multiplied by **(c)**, and finally, the result is multiplied by the result of the previous sum. All that is mathematically expressed in only one final gate. We can represent layers and layers of these operations, reducing them all to one arithmetic circuit.The second step is a **Rank 1 Constraint System** (**R1CS**) representation. In R1CS, we have a group of three vectors (**A**, **B**, and **C**), as you can see in the following figure. The solution to satisfy the system is a new vector **(S)** given by an operation of a **(.)** dot product between the vectors, of which the final result has to be zero. So, R1CS has this scheme of operations and must satisfy the following equation with a result of zero: `(S • A) * (S • B) - (S • C) = 0` For example, this is a satisfied R1CS system:

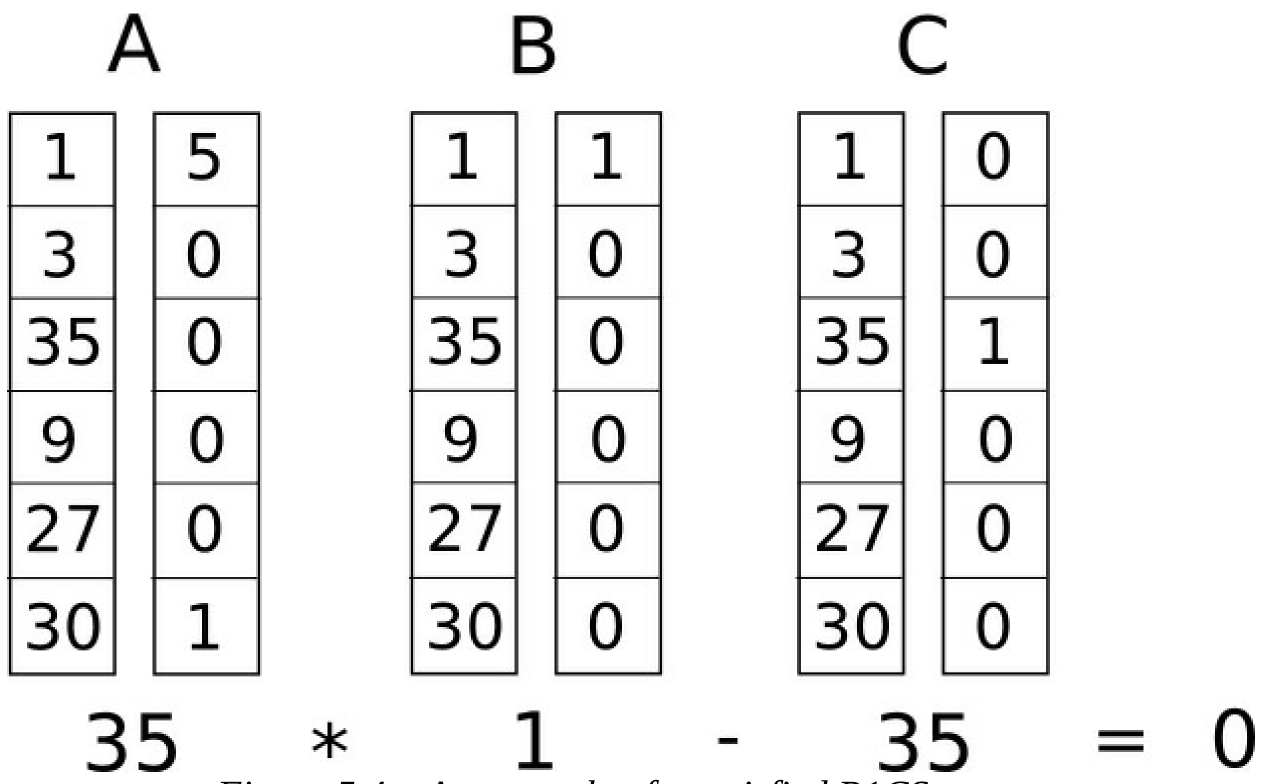


Figure 5.4 – An example of a satisfied R1CS system

As you can see in *Figure 5.4*, the value of the vector **[S]** is **[1, 3, 35, 9, 27, 30]**, which ensures a satisfied R1CS system. Indeed, if you look at column **[A]**, the result of the operations (**.**) at the end of the column is as follows: `[1.5 + 3.0 + 35.0 + 9.0 + 27.0 + 30.1] = [5 + 0 + 0 + 0 +`
result of the operations in column **[B]** is as follows:

```
[1.1 + 3.0 + 35.0 + 9.0 + 27.0 + 30.0] = [1 + 0 + 0 + 0 + 0 + 0]
```

the result of the operations in column **[C]** is as follows:

```
[1.0 + 3.0 + 35.1 + 9.0 + 27.0 + 30.0] = [0 + 0 + 35 + 0 + 0 + 0]
```

R1CS checks whether the values are *traveling correctly*. It's a verification of the values. In our example in *Figure 5.4* for instance, R1CS will confirm that the value coming out of the multiplication gate where (**b**) and (**c**) went in is (**b*c**).In the third step, Zcash converts R1CS *flat code* to a **Quadratic Arithmetic Program** (**QAP**), which operates on polynomials in (**mod x**).So, the next step is taking R1CS and converting it into QAP form, which implements the same logic as before, except using polynomials instead of dot (**.**) products between vectors. As I told you before, I will limit myself to explaining the Zcash process at a high level, so I will not go any deeper in analyzing the third step of the QAP. At this point, can you guess why the inventors of Zcash put so much effort into this system? It is probably because the inventors aspired to create the *perfect* ZKP. Indeed, in the paper entitled *Aurora: Transparent Succinct Arguments for R1CS*, the authors stated that their goal is to obtain transparent zk-SNARKs that satisfy the following conditions:

- **Post-quantum security**: *This is motivated by the desire to ensure the long-term security of deployed systems and protocols.*
- **Concrete efficiency**: *We seek argument systems that not only exhibit good asymptotics (in argument size and prover/verifier time), but also demonstrably offer good efficiency via a prototype.*

Given the high expectations predicted by the authors for this protocol, let's go on to explore the final steps of the sequence. This protocol offers a probabilistic solution by performing a multiplication of polynomials. If the two polynomials match at a random point, we can be confident that the chosen point verifies the proof correctly. The reason for this transformation is that instead of checking the constraints individually, as in R1CS, we can now check all the constraints at the same time. Here, you can see an example of how the vector verification looks in a QAP:

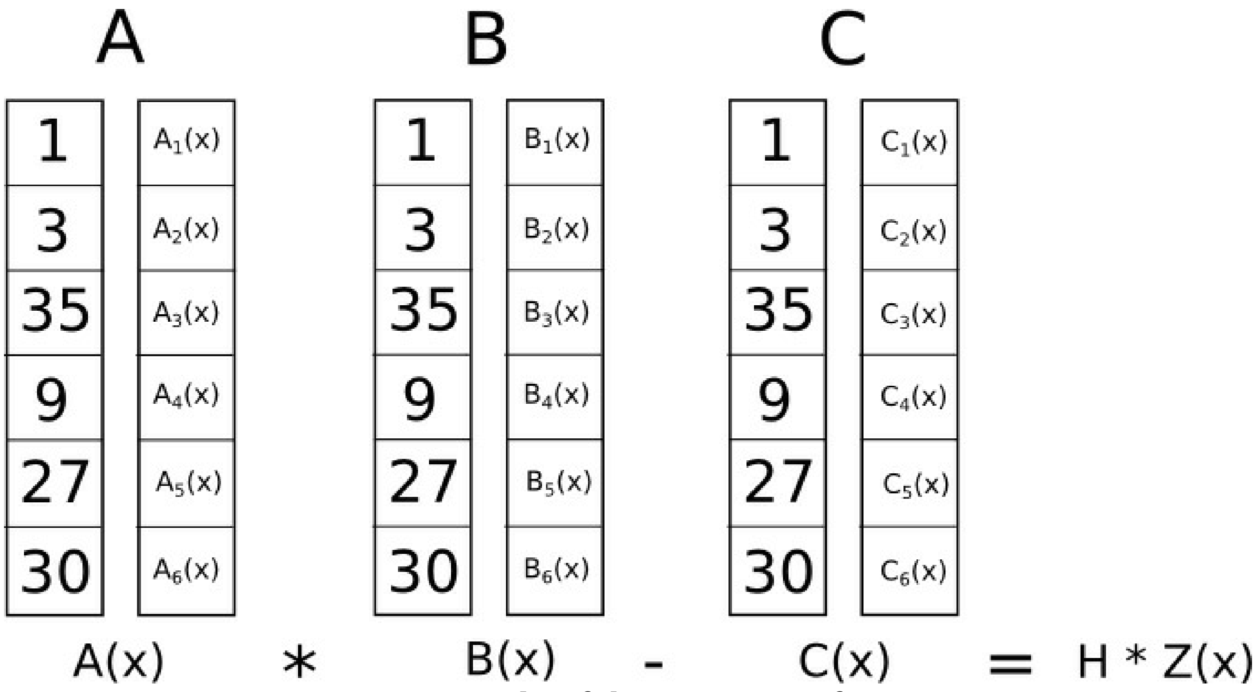


Figure 5.5 – An example of the vector verification in a QAP

Can you recognize the differences between this representation and the *checksum* in R1CS?In both cases, if the logic gate is equal to zero, the result given by the dot (**.**) product of the checks passes; if at least one of the (**x**) coordinates gives a non-zero result, this means that the values going into and out of that logic gate are inconsistent.But at this point, there could be a problem: if someone knows in advance which point the verifier chooses to check the validity, they can generate an invalid polynomial, but it could still satisfy that point.Essentially, this is a dangerous step and could be vulnerable to attacks. To overcome this problem, Zcash applied sophisticated techniques to zk-SNARKs in order to evaluate the polynomials *blindly*. These mathematical techniques used in Zcash, such as homomorphic encryption and pairings of elliptic curves, help to blind the operations but increase complexity. We will look at these issues in the next section, but now we will finish discussing the entire process of zk-SNARKs in Zcash. As I said at the beginning of the Zcash explanation, the goal of the protocol is to determine whether a statement (in this case, a transaction) is true or false, thereby preventing the *double-spending* problem.

Conclusions and the weaknesses of zk-SNARKs in Zcash

As we saw in the previous section, one of the weak points of this protocol can be found in QAP. As I have explained, Zcash has tried to overcome this problem using homomorphic evaluation, in other words, keeping the polynomials *in blind*. The issue is that homomorphic encryption usually causes bit-overflow; moreover, the protocols and schemes required to achieve fully homomorphic encryption are very complex. As you already know, my theory is that *in cryptography, complexity is the enemy of security*. I won't enter this debate now because it's not within the scope of the book to analyze the entire protocol of Zcash. Imagine the scenario discussed in the *Non-interactive ZKPs* section based on the RSA problem. If I have to demonstrate to an expert that I hold the formula for an atomic bomb, then the experts will probably ask me to show something more than a hash function of the document, **[m]**, that states the proof. The verifier will be convinced only when they get substantial proof. In other words, ZKPs are limited in the amount of evidence of knowledge they are able to provide.

One-round ZKP

In this section, we'll explore a little-known ZKP composed of only *one round* of encryption that was presented by two researchers, Sultan Almuhammadi and Clifford Neuman, of the University of Southern California, which purports to give proof of knowledge for a challenge in just one round. The paper states the following: *"The proposed approach creates new protocols that allow the prover to prove knowledge of a secret without revealing it."* The researchers also proved that a non-interactive ZKP is more efficient in terms of computational and communications costs because it saves execution time and reduces latency in communication. ZKPs are used in many fields of information technology, such as e-commerce applications, smart cards, digital cash, anonymous communication, and electronic voting. Almuhammadi and Neuman sought to satisfy the requirements of ZKPs but in just one round, eliminating any iterative mathematical scheme that would entail high computation and communication costs.So, let's dive deep to analyze this one-round ZKP and see how it works.Let's say that Peggy wants to demonstrate to Victor that she knows a discrete logarithm (we'll be focusing on a discrete logarithm, but the protocol can work for other problems); in order to do this, Peggy has to demonstrate that she knows **[x]**, as follows: `g^[x] ≡ b (mod p)` Victor launches a challenge (**c**) to verify whether Peggy really knows **[x]**. He picks up a random **[y]** and calculates the preceding function: `c ≡ g^[y] (mod p)` Victor sends (**c**) to Peggy. She inserts the parameter **[x]** on (**c**), computing (**r**) as follows: `c^[x] ≡ r (mod p)` Peggy sends (**r**) to Victor, who can verify the following: `r ≡ b^[y] (mod p)` Finally, Victor accepts the verification if **(r)** corresponds to **V = b^[y] (mod p)**.This protocol looks very simple and straightforward. It is based on the computational difficulty to calculate the discrete logarithm, as you have seen in previous cases. But to help you better understand the operations, I will show how it works mathematically and demonstrate a numerical example in the next section.

How it works mathematically

The first question is: why are the parameters (**r** and **V**) mathematically identical?Here, you can find the answer: `r ≡ c^[x] ≡ g^[y]^[x] ≡ g^[y*x] (mod p)V ≡ b^[y] ≡ g^[x]^[` you can see, **r ≡ V ≡ b^y (mod p)**.**Numerical example**Let's look at a numerical example: `p = 2741g = 7` **x = 88** is the secret number that Peggy has to demonstrate that she knows.The statement is as follows: `g^x ≡ b (mod p)7^88 ≡ 1095 (mod 2741)b = 1095` Victor chooses a random number: `y = 67` Victor calculates the following: `g^y ≡ c (mod p)]7^67 ≡ 1298 (mod 2741)c = 1298` Peggy, after having received (**c**), calculates the following: `c^x ≡ r (mod p)1298^88 ≡ 361 (mod 2741)r = 361` Peggy sends (**r**) to Victor, who can verify the following: `b^y ≡ V (mod p)1095^67 ≡ 361 (mod 2741)V = 361 = r` Finally it is verified! As you can see, we have proved the one-round ZKP with the help of a numerical example. In the next section, I will demonstrate the strong similarity of this protocol with a protocol we have analyzed before in this book: **Diffie-Hellmann** (**D-H**).

Notes on the one-round protocol

Having analyzed this protocol, you may have noticed that it is similar to the D-H exchange. Undoubtedly, the authors of the one-round ZKP were well aware of that. Still, even though the aim of the one-round ZKP is different from that of D-H, I will compare the two algorithms so that you can see what similarities there are. In the second part of the analysis, we will see how efficient this protocol is. Indeed, with only two steps, Peggy can demonstrate to Victor that the statement **[x]** is valid.Now, we can reassemble the one-round protocol using the following method: **Step 1**: Peggy `g^x ≡ b (mod p)` That is the same in D-H as the following: `g^a ≡ A (mod p)` **Step 2**: Victor `g^y ≡ c (mod p)` That is the same in D-H as the following: `g^b ≡ B (mod p)` **Step 3**: Peggy `c^x ≡ r (mod p)` In D-H, this becomes the shared key H: `B^a ≡ H (mod p)` **Step 4**: Victor `b^y ≡ r (mod p)` In D-H, this again becomes the shared key H: `A^b ≡ H (mod p)` So, **[H]** is the shared private key that "Alice and Bob" (here, Peggy and Victor) use to compute in D-H. Here, it is just **[r = H]** that gives the proof to Victor.So, we can certainly say that Sultan and Clifford's protocol is identical to D-H, as discussed in *Chapter 3, Asymmetric Encryption*.This protocol undoubtedly verifies that Peggy knows **[x]**. She can demonstrate it to Victor even if Victor doesn't know **[x]**. That is the exciting point, and the innovation of this protocol: even if Victor doesn't know **[x]**, by using this protocol, he can be confident that Peggy knows it. In other words, what the authors of this protocol did was apply the D-H protocol to the ZKP use case. If you look at the simplified version of the protocol shown below, you will get an even better understanding of the steps required. There are only two, essentially:

- Initialization of the parameters for Peggy is **g**, **b**, **p**, and **x**.
- Victor generates a random **y**.

Step 1: Victor sends the following to Peggy: `c ≡ g^y (mod p)` **Step 2**: Peggy sends the following to Victor: `r ≡ c^x (mod p)` Instantly, Victor can verify the following: `r ≡ b^y (mod p)` As you can see, there are only two steps required to perform this protocol and verify the statement **[x]** through Victor's validation of the parameter (**r**). This protocol gave me the inspiration to build a new protocol, which we will explore in the next section. My research has allowed me to reduce the number of steps to one.

A new Algorithm proposed by the Author: ZK13 – a ZKP for authentication and key exchange

The **ZK13** protocol was invented and patented by me in 2013. It's a non-interactive protocol that solves an issue that was very important to my **Crypto Search Engine** (**CSE**) project (explained in Chapter 9): authentication without the need of a public key. In this section, we will analyze this ZKP that's used for authentication. It doesn't matter whether it's humans or computers; we could call this protocol a **ZK-proof of authentication**. To better understand the problem, imagine Alice and Bob want to share a common secret, something that only they know. Let's say that the secret is the answer to the following question: how many birds were counted at the lake shoreline today? The answer is known only to Alice and Bob, unless they have revealed it to someone, but this is a problem we will take into consideration later. For now, nobody else can know the answer except Alice and Bob. We can consider the number of birds counted as a shared secret, a key that doesn't need to be exchanged. It is shared by Alice and Bob, a key that is implicitly formed by a common experience. So, besides the authentication problem, there is also the problem of verifying a private **Pre-Shared Key** (**PSK**). Indeed, under ZK13, Alice tells Bob to use the secret shared key (the number of birds counted) as a secret password, **[private key]**. What's even more interesting here (and this is where it really differs from the D-H key exchange algorithm we saw before) is that the secret key is not *really* exchanged at all, but instead is simply something that is known to both parties and is only verified.So, the problems that this ZKP can solve are several. Here, we will just consider the authentication problem. Later in the book, we will analyze how to use a ZKP to exchange a shared private key. In 2013, I was drawing up the architecture of the CSE, a project that has absorbed a large part of my professional life We will talk in detail about the CSE in *Chapter 9, Crypto Search Engine*. At the time, one of the toughest problems to solve with the CSE's architecture was finding a cryptographic method to identify the **Virtual Machines** (**VMs**) network. Since the algorithm chosen was symmetric, the problem was to find a method of authentication that would work with the symmetric algorithm. As you already know from *Chapter 2, Introduction to Symmetric Encryption*, it's common to think that symmetric algorithms don't have a digital signature method of authentication because they do not have public keys. At first glance, it doesn't seem easy to find such a method of authentication, but it can be possible if the process starts with a shared secret. The goal was to prevent a man-in-the-middle attack by an external hostile VM against a network.To overcome all these issues, I considered implementing a new ZKP. Taking a look at the most popular ZKPs, I did consider Schnorr (presented earlier in this chapter) as a candidate. But an *interactive protocol* didn't fit well. This scheme needs more steps between the prover and verifier, generating latency in the communication. So, I decided to implement a new *personal* zero-knowledge non-interactive protocol.After many studies and a pinch of inventiveness, I designed ZK13. Before analyzing it, I will explain what constraints I worked under:

- The secret shared key (the challenge) has to be embedded inside the VM database. Therefore, engineers could *inject* the secret parameter **[s]** into both of the VMs without exchanging any keys through an asymmetric algorithm.
- The goal of ZK13 is to enable parties to identify each other by sharing a small amount of sensitive information. This means exchanging only the minimum amount of sensitive information (that is, the hash of **[m]**:

H(m) instead of **[m]** itself) that needs to be shared. Indeed, the greater the amount of information exchanged, the greater the vulnerability to attack becomes.

- ZK13 had to be a simple and, as I have already said, *non-interactive* protocol. Therefore, only one piece of information should be required by the prover. The reasons for this are twofold: first, to avoid an excess of information being exchanged (see the previous point) because that could compromise security. The second reason is related to the goal of the application: the CSE is a platform on which encrypted data is searched and retrieved using the cloud or external servers. Because a search engine has to be fast, queries should be fielded and answers given in the least amount of time possible. So, it is crucial to avoid latency during the authentication phase.
- Another constraint of ZK13 was for it to use the best and most secure authentication methods. At the time that it was conceived (2011–2013), the quantum computing era was not yet seen as dangerous for cryptography. So, the underlying problem on which the system relied was the discrete logarithm, which is still considered a hard problem.

ZK13 explained

The ZK13 protocol, with only one transmission and a shared secret, is presented as follows:

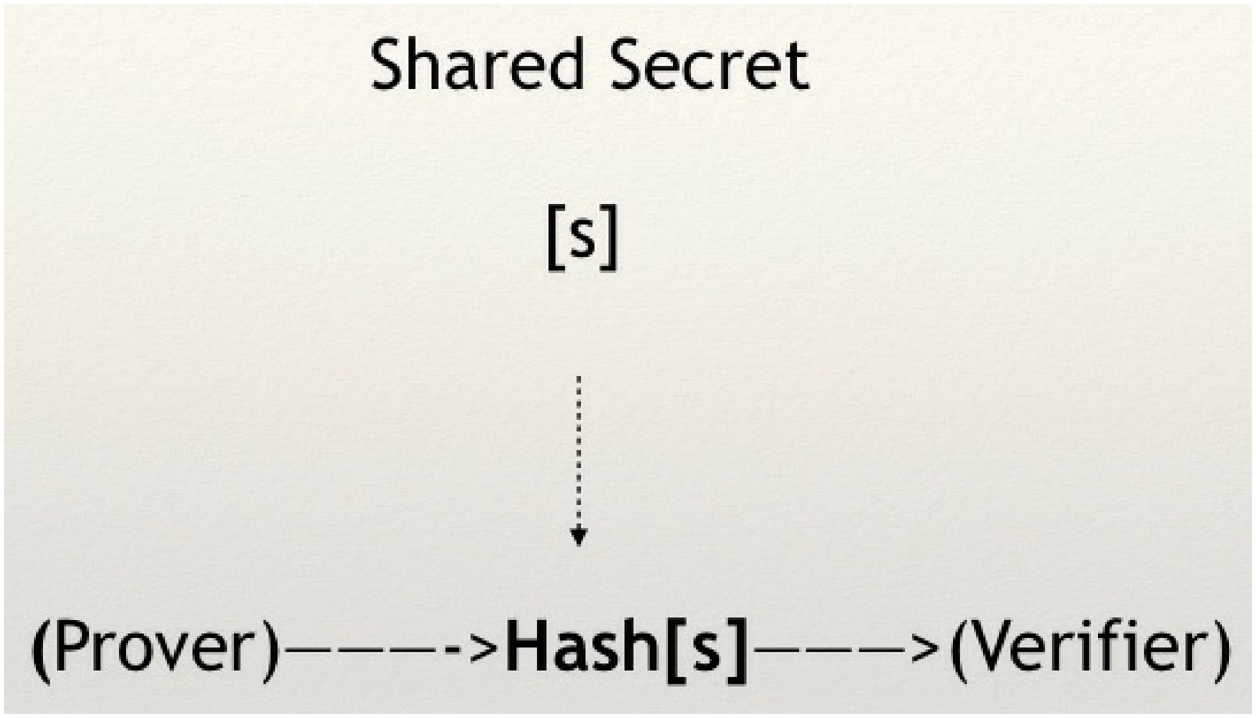


Figure 5.6 – The scheme of the shared hash[s] secret

Let's dive deeper into ZK13 and see how it works.Bob (VM-1) has to prove that he knows the secret, **[s]**, to Alice (VM-2) in order to send Alice a set of encrypted files using the CSE system. Remember that **[s]** is stored inside the *brains* of both Alice and Bob, the two VMs, as an innate native injected secret.Bob picks a random number **[k]** (the (**G**) element of a zk-SNARK or the random key generator). This random number, **[k]**, is generated and destroyed in each session:**Public parameters**:**p**: This is a prime number. **g**: This is the generator. **Key initialization**:**[k]**: This is Bob's random key. **Secret parameters**:**[s]**: This is the common shared secret.**H[s]**: This is the hash of the secret, **[s]**.**Step 1a**: Bob calculates (**r**) as follows: `r ≡ gk (mod p) r ≡ g^k (mod p)` Let's say that the secret shared is **[s]**, but effectively, the VM operates with **H[s]**, the hash functions of **[s]**.**Step 1b**: Bob calculates **[F]**, a secret parameter, which is changed in each session (just because **[k]** changes): `H[s]*k ≡ F (mod p-1)` Now, with (**g**) raised to **[F]**, Bob proves (**P**), which is the second element of the zk-SNARK: `gF ≡ P (mod p)` Bob sends the pair (**P**, **r**) to Alice.**Step 2**: The verification step (**V**) validates the prove, (**P**), based on the function: `[s] —> H[s]r[Hs] ≡ gF (mod p)` If: `V ≡ r[Hs] = P (mod p)` Alice proceeds to make a hash of **[s]: H[s]**, and then she accepts the authentication if **V = P**; if ((**V**) is not equal to (**P**)), she doesn't accept the validation.In this case, as we have supposed in the initial conditions, **[s]** is supposed to be known by Alice.As you can see, ZK13 works in only two steps, but the verifier (in this case, Alice) must know the secret, **[s]**; otherwise, it is impossible to verify the proof.**Numerical example**:Now, let's see a numerical example of the ZK13 protocol:**Public parameters**: `p = 2741g = 7` **Secret parameters**: `H[s] = 88k = 23 g^k ≡ r (mod p)7^23 ≡ 2379 (mod 2741)r` Bob calculates **[F]** and then (**P**): `[Hs] * k ≡ F (mod p-1)88 * 23 ≡ 2024 (mod 2741-1)F = 2024g^F ≡` verifies the following: `r^[Hs] ≡ P (mod p)2379^88 ≡ 132 (mod 2741)` Alice double-checks whether **[Hs] = [s]**; if it's **TRUE**, then it means that Bob does know the secret, **[s]**. Now that we have proven that ZK13 works with a numerical example, I want to demonstrate how it works mathematically.

Demonstrating the ZK13 protocol

Since **P ≡ g^F (mod p)**, what we want to demonstrate is the following: `P ≡ gF ≡ rs (mod p)` (Here, I use **[s]** for the demonstration instead of **H[s]**.)As **r ≡ g^k (mod p)**, substituting (**r**) in the preceding equation, we have the following: `P ≡ gF ≡ (g^k)^s (mod p)` We also know that **F** is the following: `F ≡ s*k (mod p-1)` Finally, for the properties of the modular powers substituting both **[F]** and (**r**), we get the following: `P ≡ gs*k ≡ (g^k)^s ≡ g^k*s (mod p)` Basically, I have substituted the parameter (**P**), the proof created by Bob, with the elements of the parameter itself, demonstrating that the secret, **[s]**, is contained inside (**P**). So, (**P**) has to match with the *ephemeral* parameter **(r)^[s]** generated by Bob and sent to Alice together with the proof, (**P**). If Alice knows **[s]**, then she can be sure that Bob also knows **[s]** because (**P**) contains **[s]**. That's what we wanted to demonstrate.

Notes and possible attacks on the ZK13 protocol

You will agree with me that using this protocol, it is possible to determine proof of knowledge of the secret, **[s]**, in only one transmission. During the explanation of the algorithm, I have divided it isnto three steps, but actually, there are only two steps (with only one transmission), because the operations of (**G**) key generation are offline. So, steps 1a and 1b can be combined into effectively only one step.

Possible attacks on ZK13

Let's say Eve (an attacker) wants to substitute herself for Alice or Bob, creating a man-in-the-middle attack.This could be done as follows.Eve replaces (**r**) with (**r1**), generating a fake (**k1**), by calculating the following: `r1 ≡ g^k1 (mod p)` But when Eve computes **[F]**, she doesn't know **H[s]** (because it's assumed that **[s]** will remain secret). So, this attack fails.Instead, she can collect (**r**, **P**) and replay these parameters in the next session, activating a so-called **replay attack**. This attack could be avoided here, because (**r**) is generated by a random **[k]**, so it is possible to avoid accepting an (**r**) already presented in a previous selection. So, that was one attack that could be faced, and we saw how to avoid it.

Summary

Now you have a clear understanding of what ZKPs are and what they are used for.In this chapter, we have analyzed in detail the different kinds of ZKPs, both interactive and non-interactive. Among these protocols, we saw a ZKP that used RSA as an underlying problem, and I proposed an original way to trick it.Then, we saw the Schnorr protocol implemented in an interactive way for authentication, on which I have proposed an attempt to attack.Moving on, we explored the zk-SNARKs protocols and *spooky moon math*, just to look at the complexity of some other problems. Among them, we saw an interesting way to attack a discrete logarithm-based zk-SNARK. We dived deep into Zcash and its protocols to see how to anonymize the transactions of this cryptocurrency. Later in the chapter, we encountered and analyzed a non-interactive protocol based on the D-H algorithm. Finally, we explored ZK13, a non-interactive protocol, and its use of shared secrets to enable the authentication of VMs.Finally, we explored the zkSNARKs in the world of Cryptocurrencies, expecialy used to anonymize the transactions.You became familiar with some schemes of attacks, such as man-in-the-middle, and used some mathematical tricks to experiment with ZKPs.The topics covered in this chapter should have helped you understand ZKPs in greater depth, and you should now be more familiar with their functions. We will see in later chapters many links back to what we explored here. Now that you have learned the fundamentals of ZKPs, in the next chapter, we will analyze some private/public key algorithms that I have invented.

7 Elliptic Curves

Elliptic curves are the new frontier for decentralized finance. Satoshi Nakamoto adopted a particular kind of elliptic curve to implement the transmission of digital currency in Bitcoin called **secp256K1**. Let's see how it works and what the main characteristics are of this very robust encryption.In this chapter, we will learn the mathematical basics of elliptic curve cryptography. This topic involves geometry, modular mathematics, digital signatures, and logic.Moreover, I will present the special kind of elliptic curve implemented for the digital signature of Bitcoin known as secp256K1. Finally, we will discuss the possibility of an attack on elliptic curves. So, in this chapter, we will cover the following topics:

- The genesis of cryptography on elliptic curves
- Mathematical and logical basics of elliptic curves
- The Diffie–Hellman key exchange based on elliptic curves
- An explanation of ECDSA on secp256K1 – the digital signature of Bitcoin
- Possible attacks on elliptic curves

Let's dive deep into this intriguing topic, based on geometry and applied in cryptography.

An overview of elliptic curves

Around 1985, Victor Miller and Neal Koblitz pioneered *elliptic curves* for cryptographic uses. Later on, Hendrik Lenstra showed us how to use them to factorize an integer number.Elliptic curves are essentially a geometrical representation of particular mathematical equations on the Cartesian plane. We will start to analyze their geometrical models in the 2D plane, conscious that their extended and deeper representation is in 3D or 4D, involving irrational and imaginary numbers. Don't worry now about these issues; it will become clearer later on in this chapter.**Elliptic Curves Cryptography** (**ECC**) is used to implement some algorithms we have seen in previous chapters, such as **RSA**, **Diffie–Hellman** (**D–H**), and **ElGamal**. Moreover, after the advent of the revolution in digital currency, a particular type of elliptic curve called secp256K1 and a digital signature algorithm called **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) have been used to apply digital signatures to Bitcoin to ensure that transactions are executed successfully.This chapter intends to take you step by step through discovering the logic behind elliptic curves and transposing this logic into digital world applications.It has been estimated that using 313-bit encryption on elliptic curves provides a similar level of security as 4,096-bit encryption in a traditional asymmetric system. Such low numbers of bits can be convenient in many implementations requiring high performance in timing and bandwidth, such as mobile applications.So, let's start to explore the basis of elliptic curves, which involve geometry, mathematics, and many logical properties that we have seen in earlier chapters of this book.

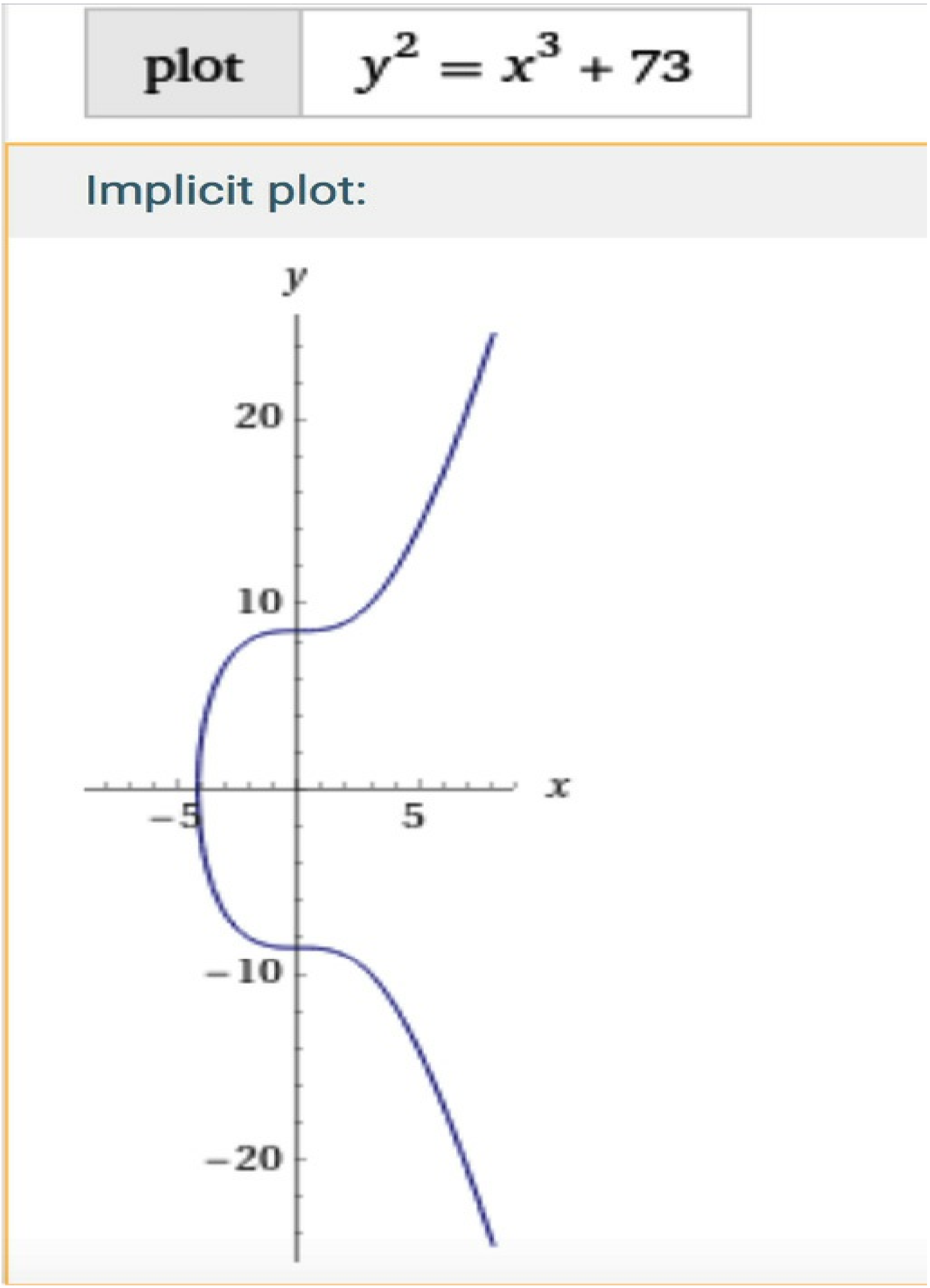
Operations on elliptic curves

The first observation is that an elliptic curve is not an *ellipse*. The general mathematical form of an elliptic curve is as follows: `E: y^2 = x^3 + ax^2 + bx + c`

Important Note

E: represents the form of the elliptic curve, and the parameters (**a**, **b**, and **c**) are coefficients of the curve.

Just to give evidence of what we are discussing, we'll try to plot the following curve: `E: y2 = x3 + 73` As we can see in the following figure, I have plotted this elliptic curve with WolframAlpha represented in its geometric form:



plot	$y^2 = x^3 + 73$

Implicit plot:

Figure 7.1 – Elliptic curve: E: y^2 = x^3 + 73

We can start to analyze geometrically and algebraically how these curves work and their prerogatives. Since they are not linear, they are easy to implement for cryptographic scopes, making them adaptable. For example, let's take the curve plotted previously: `E : y^2 = x^3 + 73` When (**y = 0**), we can see that, geometrically, the curve intersects the *x* axis at the point corresponding to (**x = -4.1793…**); this is mathematically given if we substitute **y = 0** into the equation: `y^2 = x^3 + 73` As a result, one of the three roots of the curve will be the cubic root of **-73**.When (**x = 0**), the curve intersects the *y* axis at two points: (**y = +/-8.544003…**). Mathematically, we substitute (**x = 0**) into the equation: `y^2 = x^3 + 73` This obtains the intersection between the curve and the axis of *y*, as you can see in *Figure 7.1*.Another curious thing about elliptic curves, in general, is that they have a point that goes to infinity if we intersect two points in the curve symmetrically with respect to the *y* axis. You can imagine the third point as an infinite point *lying* at the infinite end of the *y* axis. We represent it with *O* (*at the infinity point*). We will see this better later when we discuss adding points to the curve.One of the most interesting properties of elliptic curves is the **SUM** value of two points of the curve. As you can see in the following figure, if we have two points, **P** and **Q**, and we want to add **P** to **Q**, it turns out that if we draw a line between **P** and **Q**, a third point, **-R**, is given by the intersection between this line and the curve. Then, if you take a reflex of point **-R** with respect to the *x*-axis line, you find **R**; this is the sum of **P** and **Q**. It's easier to take a look at the following diagram to understand this geometrical representation of the **SUM** value:

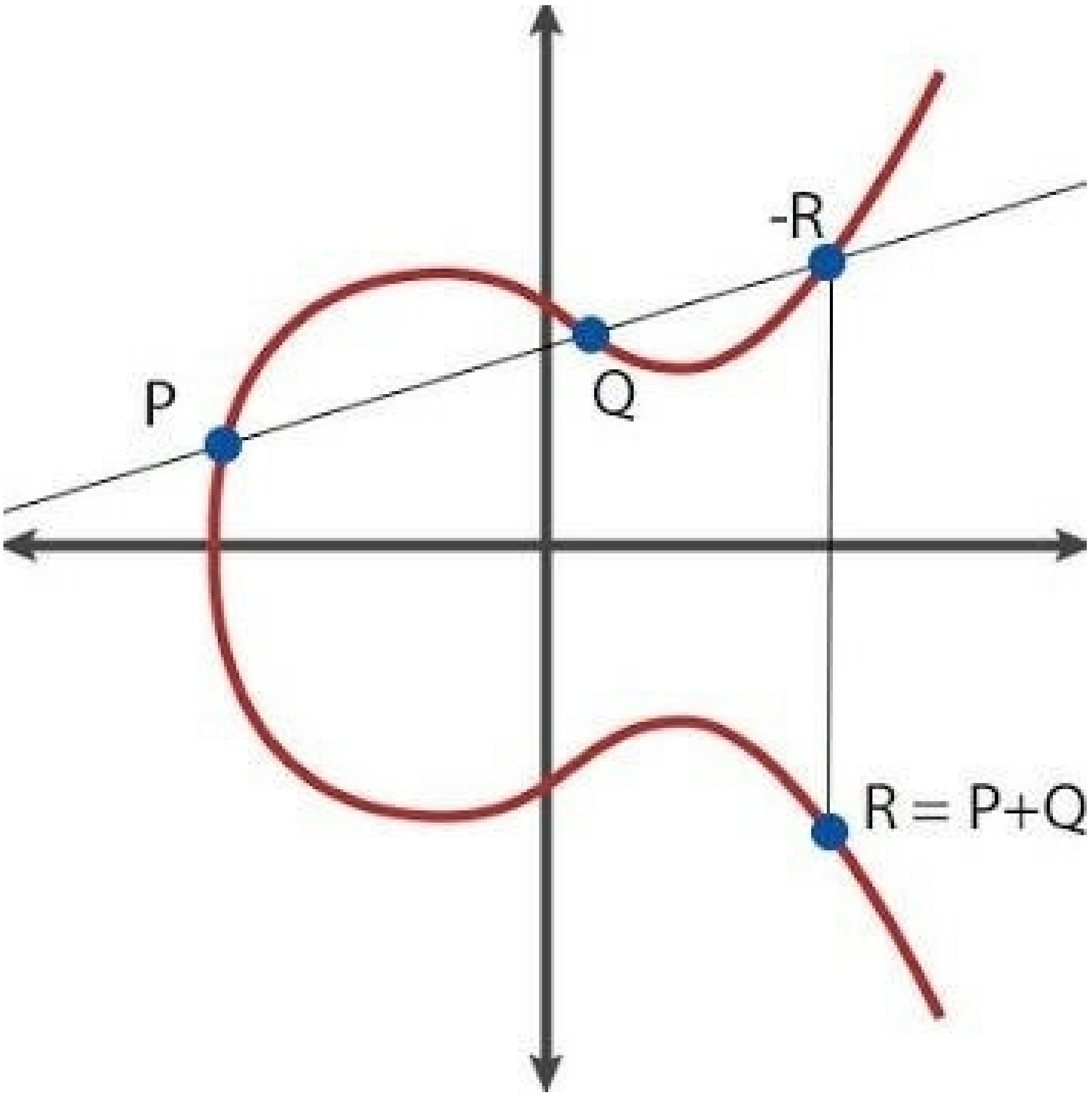


Figure 7.2 – Adding and doubling points on the elliptic curve

Now, let's look at how the addition point will be represented algebraically.First, we take the coordinates of **P** and **Q**, and then calculate (**s**), as follows: `s = (yP - yQ)/(xP - xQ)` To compute **xR**, the *x* coordinate of point **R**, we have to perform this operation: `xR = s^2 - (xP + xQ)` To compute **yR**, the *y* coordinate of point **R**, we have to perform this operation: `yR = s(xP - xR) - yP` What about computing **P + P = R = 2P**, the so-called **point double**?If we want to represent a **P** point added to itself so that it becomes **2P**, the geometrical representation is given by the tangent passing through the **P** point and intersecting the curve at the **R** point, and again finding the reflexed point of the sum that is the symmetric point **R**, as shown here:

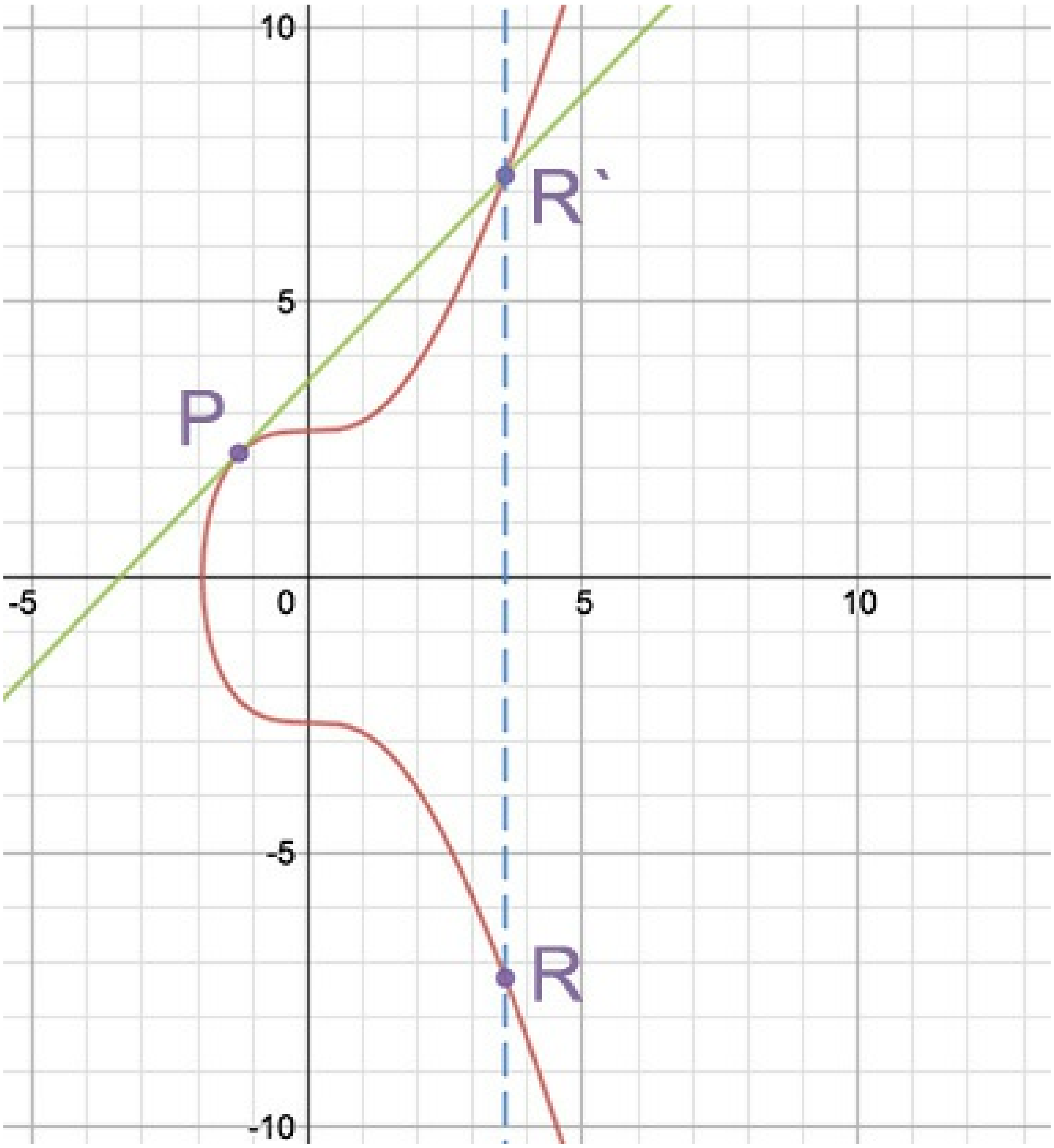


Figure 7.3 – 2P point double

Geographically, it is very similar to computing the sum of the point, as we saw in the preceding example.We have to draw the tangent line in the **P** point, and we find the point of intersection between the line and the curve in the **R'=(-2P)** point; finally, the reflexed *x*-axis point on the curve will give the **P** double point, that is, **R (2P)**.What about computing **P+P** algebraically? In this case, (**t**) will be the following: `t = (3XP^2 + a)/2YP` Remember that (**a**) is a parameter of the curve.So, to find the *x* coordinate of **R**, we have the following: `xR = t^2 - 2XP` The equation for the *y* coordinate of **R** is as follows: `yR = t (xP - xR) - yP` There is one more case we need to address when we operate with elliptic curves: how to add vertical points.Here is **O** represented by the *point of infinity*, given by the **SUM** value between **P** and **Q** if **xP = xQ**:

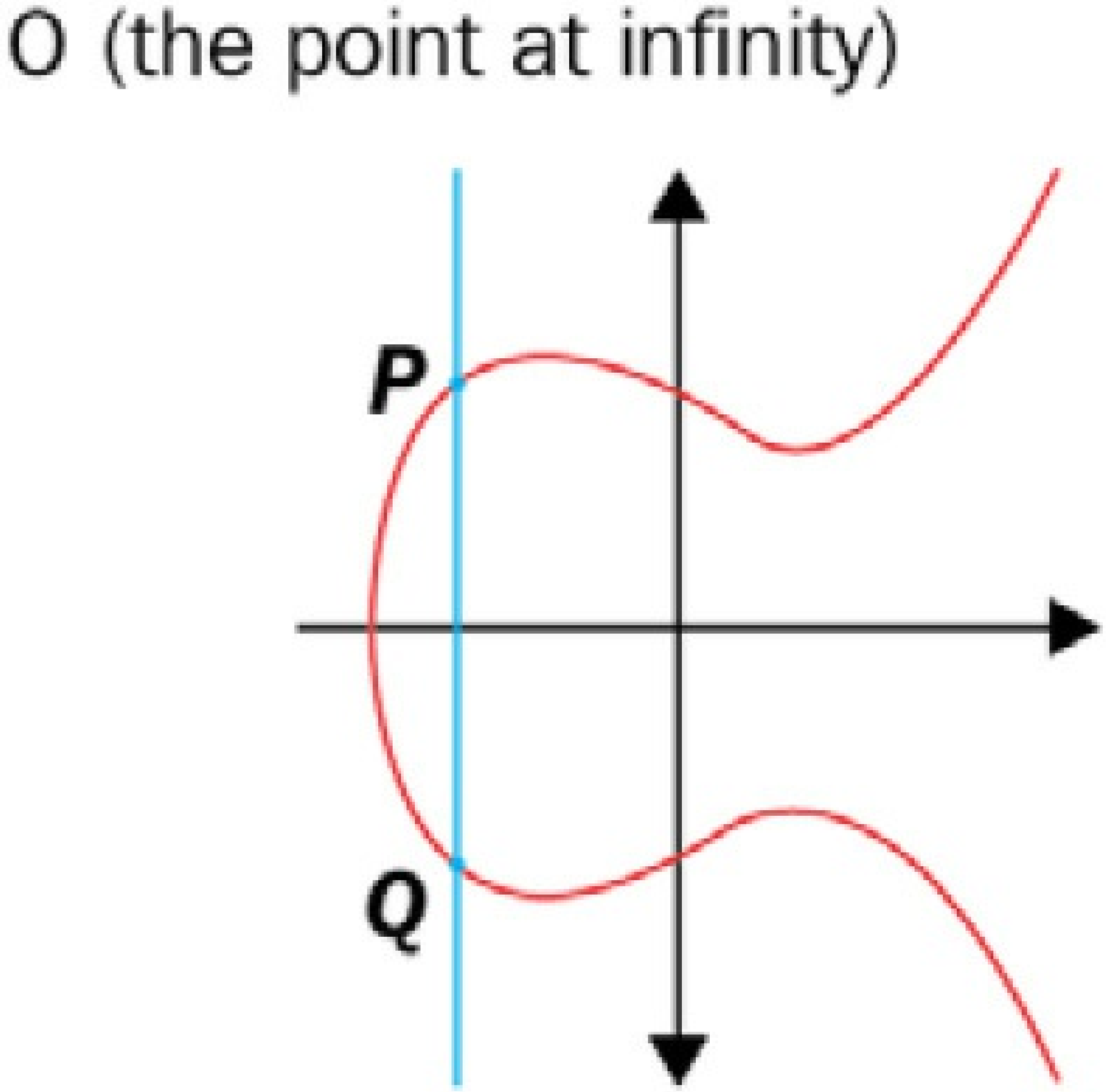


Figure 7.4 – Adding vertical points

Algebraically, the point at infinity is given as follows: `P + Q = O (point at infinity), if xP = xQ` Alternatively, it can be given like this: `P + P = O (point at infinity), if xP = 0` If we assume that **A** and **B** are the two points to add, we can summarize all the expressions into the following representation:

$A = (x_A; y_A) \qquad B = (x_B; y_B)$

$$x_C = \left(\frac{y_B - y_A}{x_B - x_A}\right)^2 - x_A - x_B$$

$$y_C = \left(\frac{y_B - y_A}{x_B - x_A}\right)(x_A - x_C) - y_A$$

2*A = A+A is obtained by the following equations:

$$x_{2A} = \left(\frac{3x^2{}_A}{2y_A}\right)^2 - 2x_A$$

$$y_{2A} = \left(\frac{3x^2{}_A}{2y_A}\right)(x_A - x_{2A}) - y_A$$

Figure 7.5 – Operations of addition and multiplication on elliptic curves

Now that we have seen adding and doubling points, let's see how to perform scalar multiplication, another important operation for cryptography on elliptic curves.

Scalar multiplication

At this point, we have to get familiar with another typical operation of elliptic curves, **scalar multiplication**. This process mathematically represents the sum between **P** and **Q** and makes it possible to calculate **2P**, **3P**, …, **nP** on elliptic curves.The logic behind scalar multiplication is not difficult, but it needs practice to become familiar with it. Practically, let's solve a multiplication on the elliptic curve of the following form: `Q = n*P` This is called **repeated addition**, and it can be represented as follows: `Q = {P + P+ P + P…+ P} n-times` As we have to add the coordinates of a point to itself, we can figure out the scalar multiplication as the sum of a point with itself **n** times, so we have, for example, the following: `P + P = 2P` This, as we have seen, is the formula of double point (geometrically represented in *Figure 7.3*) transposed into an algebraic representation of **SUM**. Always remember that we are dealing with a curve, so the coordinates of the new intersection point are as follows: `2P (X2P, Y2P)` For the same logic, we can go on with **3P**, **4P**, … **7P**: `R = 3PR = 2P +P` Then we follow with **4P**: `R = 4PR = 2P + 2P` Then we follow with **5P**: `R = 5PR = 2p + 2P + P` Then we follow with **6P**: `R = 6PR = 2P + 2P +2P` Alternatively, it can be given like this: `R = 2(3P)R = 2 (2P+ P)` Finally, we have the following: `R = 7PR = P + (P + (P + (P + (P + (P + (P))))))` Alternativ it can be given like this: `R = 7PR = P + 6PR = P + 2(3P)R = P + 2(P + 2P)` It is called multiplication, but in reality, scalar multiplication is more of a *breakdown* of the **R** number in a scalar mode because, as you can see, step by step, it is going to be reduced to the minimal entity of **P**: so, for instance, **7P** becomes a product of **P** and **2P**.Following this logic, if we have to multiply a **K** number with **P**, we have to break it down to obtain a sum of minimal elements (**P+P**) or a **2P** double point that will compose the multiplication required. Because of the formulas, we rely on addition and multiplication, as we saw in *Figure 7.5*.If we have **9P**, for example, we will have the following: `9P = 2(3P) + 3P9P = 2(2P + P) + 2P + P9P = P + 2P + 2(2P+ P)` Now that we have understood this operation's logic, let's discover why these operations are so important to generate a cryptographic system on elliptic curves.As we saw in the previous chapters (for example, D–H in *Chapter 3, Asymmetric Encryption*) about cryptography, we are looking for one-way functions. These particular functions allow it to be easy to compute in one direction but are very difficult to perform in the opposite sense. In other words, it isn't easy to reverse-engineer the result. Similarly, in elliptic curves, we have to find a function such as a discrete logarithm that allows us to perform a *one-way* function.We'll now define the discrete logarithm problem transposed on an elliptic curve.It's stated that *scalar multiplication* is a one-way function.Given an elliptic curve: **E**.Known: **P**, **Q** is a multiple of **P**.Find **k**: Such that **Q = k * P**.It is a *hard problem to solve*. This is called a discrete logarithm problem of **Q** to the **P** base, and it is considered hard to solve because to find **k** means to calculate very complex operations. I will proceed with an example to better understand what we are dealing with.**Example**: In the elliptic curve group defined by the following, what is the discrete logarithm **k** of **Q = (4,5)** to the **P = (16,5)** base?

`y2 = x3 + 9x + 17 over the field F23`. One (naïve) way to find **k** is to compute multiples of **P** until **Q**. The first few multiples of **P** are as follows:

```
P = (16,5) 2P = (20,20) 3P = (14,14) 4P = (19,20) 5P = (13,10) 6P
```

9P = (4,5) = Q, the discrete logarithm of **Q** to the **P** base is **k = 9**. In a real application, **k** would be large enough that it would be infeasible to determine **k** in this manner. This is the fundamental principle behind the D-H algorithm implemented on elliptic curves that we will discover next.

Implementing the D-H algorithm on elliptic curves

In this section, we will implement the D–H algorithm on elliptic curves. We saw the D–H algorithm in *Chapter 3, Asymmetric Encryption*. You should remember that the problem underlying the D–H key exchange is the discrete logarithm. Here, we will demonstrate that the discrete logarithm problem could be transposed on elliptic curves too.First of all, we are dealing with an elliptic curve (mod **p**). The **base point** or **generator point** is the first element in the D–H original algorithm represented by (**g**), and here we denote it by (**G**). Let's look at some elements to take into consideration:

- **G**: This is a point on the curve that generates a cyclic group.

Cyclic group means that each point on the curve is generated by a repeated addition (we have seen point addition in the previous section).

- Another concept is the **order of G** denoted by (**n**):

`ord(G) = n` The order of (**G**) = (**n**) is the size of the group.The order (**n**) is also the smallest positive integer **[k]**, giving us the following: `kG = O (Infinity Point)`

- The next element to take into consideration is the **h** cofactor:

h = (number of points on E)/**n**In other words, (**h**) can be defined as the number of points on **E (mod p)** divided by the order of the curve (**n**). **h = 1** is optimal. If **h > 4**, the curve is more vulnerable to attacks.Let's now analyze step by step how D–H transposed on **E** works.**Step 1**: **Parameter initialization**:Now let's look at the public shared parameters to initialize the D–H model on **E (mod p)**: `{p, a, b, G, n, h}` **p** is the (mod **p**) as we have already seen in the original D–H algorithm.**a** and **b** are the parameters of the curve.**G** is the generator.**n** is the order of **G**.**h** is the cofactor.**Step 2**: **Crafting the shared key on E (mod p): [K]** After the parameter initialization, Alice and Bob will define the type of the curve to adopt among the family of **E (mod p)**: `y^2 = x^3 + ax + b (mod p)` Bob picks up a **[β]** private key such that **1≤ [β] ≤ n-1**.Alice picks up a **[α]** private key such that **1≤ [α] ≤ n-1**.After choosing random keys, Bob and Alice can compute their public keys.Bob computes the following: `B = [β] * G` Alice computes the following: `A = [α] * G` Now (like in the original D–H algorithm), there is a generation of shared keys exchanging the public parameters:

- Bob sends **B** (**xB** and **yB**) to Alice.
- Alice receives **B**.
- Alice sends **A** (**xA** and **yA**) to Bob.
- Bob receives **A**.
- Bob computes the following:

```
K = [β]*A
```

- Alice computes the following:

```
K = [α] * B
```

- Finally, Bob and Alice hold the same information: the point on **E: [K]**.

This point, **[K]**, is the shared key between Alice and Bob.

Important Note

If Eve (the attacker) wants to know **[K]**, she has to know **[α]** or **[β]**, the private keys of Alice and Bob, given by the following functions:

B = [β]* G or **A = [α]* G**

To recover **[K]**, Eve has to be able to solve the discrete logarithm on **E (mod p)**, which, as we have seen, is a hard problem to solve.

A numerical example is as follows:Let's assume that the domain of the curve is the following: `E: y^2 = x^3 + 2x +2 (mod 17)` You can take a look at this **E** plotted in *Figure 7.6*.

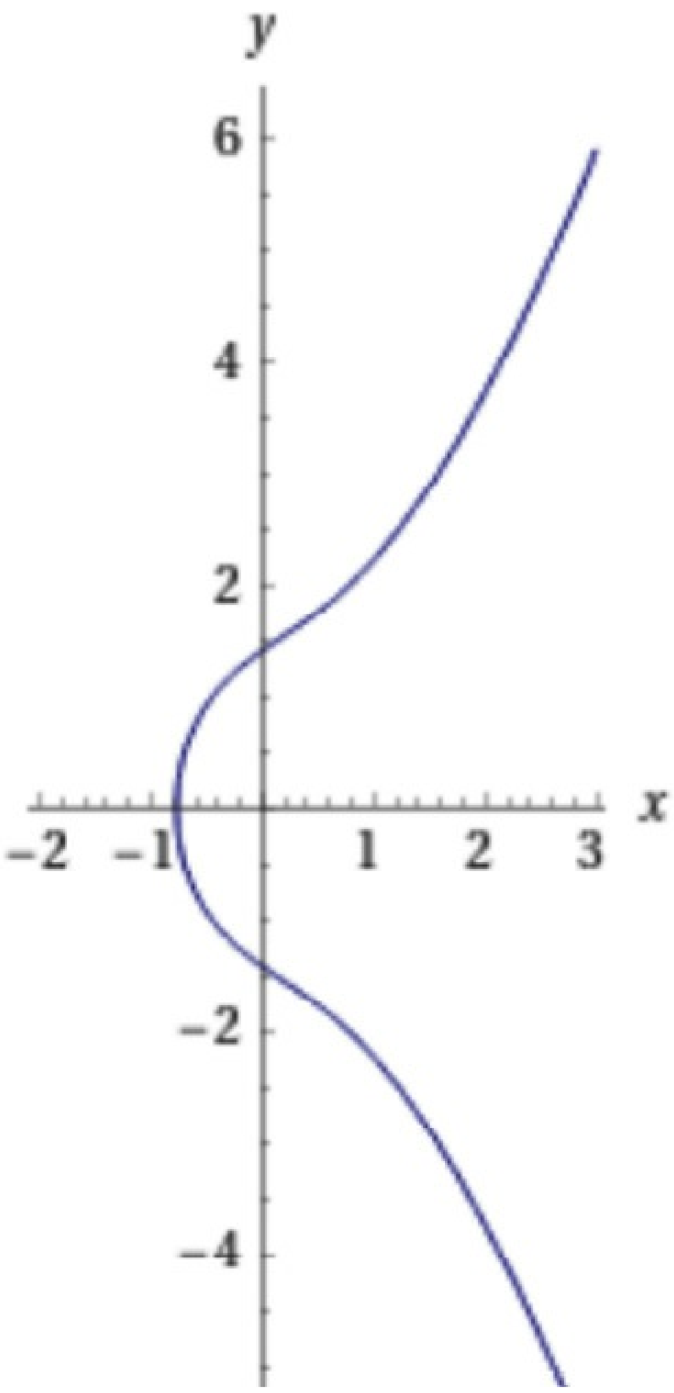


Figure 7.6 – The plotted curve used as an example to implement the D–H algorithm

The generator point is the following: `G (5,1)` First of all, we have to find the order of the **n** curve.To do that, we should calculate all the points until we reach the minimum integer that allows the cycling group.So, we start to calculate **2G**.Using the formula of the double point, we have the following: `t = (3XP^2 + a)/2YPt = (3*5^2 + 2)/2*1 = 77 * 2^-1 = Red` that we have calculated **(t)**, it's possible to use it to find the *x* and *y* coordinates on the **2G** curve:

` x2G = s^2 -2xG = 13^ 2- 2*5= Mod[13^2 - 2*5, 17] = 6 (mod 17)y2G` we found the coordinates of **2G**: `2G = (6,3)` Now, we should go on to compute **3G**, **4G**, …, **nG** until we find the point of infinity, **OG**.It is a lot of work to do it by hand, but it is also a good exercise to practice by yourself.I will let you calculate all the scalar multiplications until the **OG** point, giving the results for some points: `G = (5,1)2G = (6,3)3G = (10,6)……9G = (7,6)10G = (7,11)` This continues until it turns out that the point of infinity, **OG**, is the following: `19G = OG` This means that the order of **G** is the following: `n = 19` So, the parameters of **E** are the following: `G = (5,1); n =19` Bob picks up **Beta**: `Beta = 9` Alice picks up **Alpha**: `Alpha = 3` Bob calculates his public key (**B**): `B = 9G = (7,6)` Alice calculates her public key (**A**): `A = 3G = (10,6)` Alice sends (**A**) to Bob: `[β]*A = 9A = 9(3G) = 8G = (13,7)` Bob sends (**B**) to Alice: `[α]*B = 3B =3*(9G)= 8G =(13,7)` So, as you can see, the shared key is **[K] = (13,7)**.The parameters used in the example are too small to be implemented in a real environment. In reality, the numbers have to be larger than the ones I have used in the preceding example. However, this algorithm is computationally easier than the original D–H one and can be used with smaller parameters and keys. However, we have to rely on curves that are well structured and architected by professional cryptographers and mathematicians.

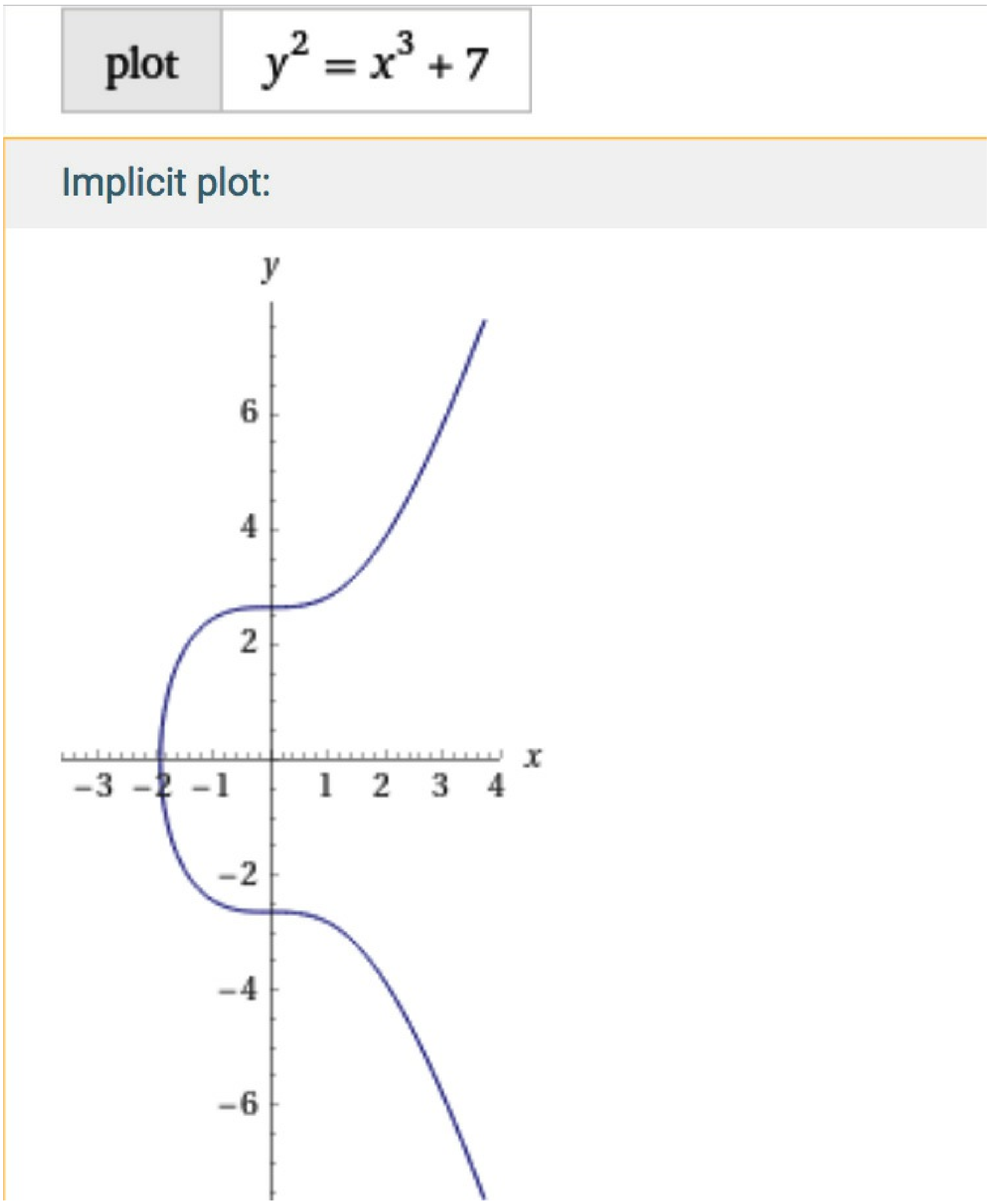
Important Note

> Implementing D–H on **E (mod p)** doesn't necessarily prevent MiM attacks, just like in the original D–H algorithm (as seen in *Chapter 3, Asymmetric Encryption*).

Now that we have more confidence with the elliptic curve and its operations, we can go through an interesting elliptic curve case, analyzing the algorithm adopted for the Bitcoin digital signature ECDSA.

Elliptic curve secp256k1 – the Bitcoin digital signature

ECDSA is the digital signature scheme used in Bitcoin architecture that adopts an elliptic curve called secp256k1, standardized by the **Standards for Efficient Cryptography Group** (**SECG**).ECDSA suggests (**a = 0**) and (**b = 7**) as parameters in the following equation: `E: y2 = x3 + 7` For a more formal presentation, you can read the document reported by the SECG at https://www.secg.org/sec2-v2.pdf, where you can find the recommended parameters for the 256 bits associated with a Koblitz curve and the other bit-length sister curves.This is the representation of secp256k1 in the real plane:


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

Implicit plot:

Figure 7.7 – secp256k1 elliptic curve

As we know, the elliptic curve has a part visible in the real plane and another representation in the imaginary plane. The form of an elliptic curve can be represented in 3D by a torus when the points are defined in a finite field, just as you can see in the following figure:

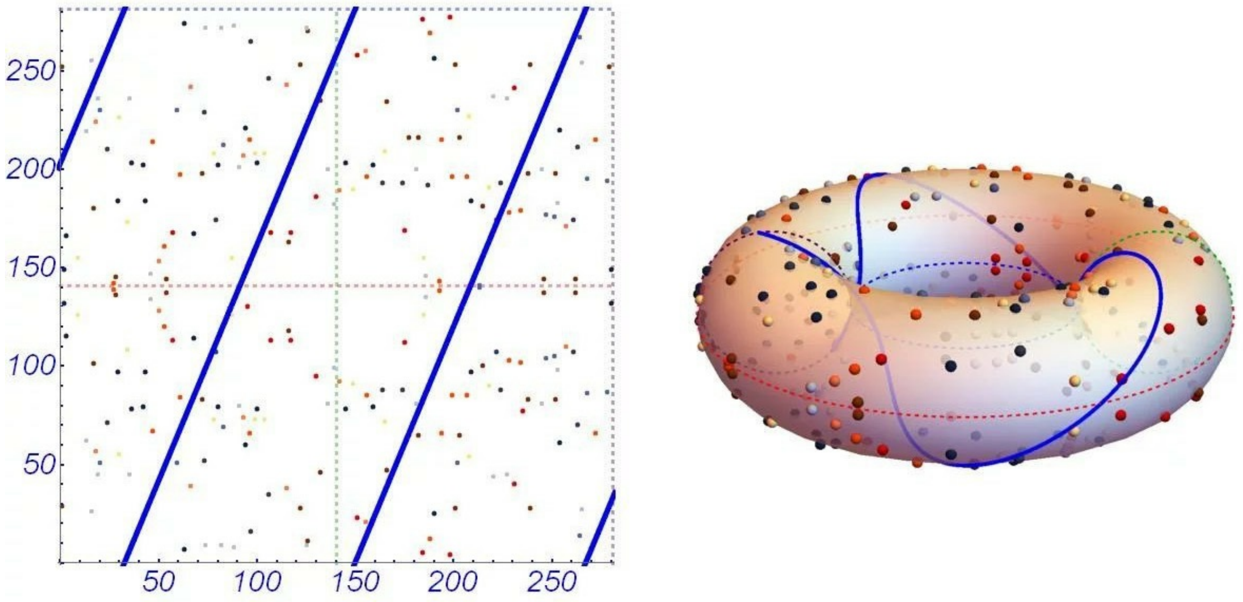


Figure 7.8 – 3D representation of an elliptic curve in a finite field

The secp256k1 curve is defined in the **Z** field as follows:
`Z modulo 2^256 - 2^32 - 977` Or, written in a different way, this is how it looks as an
integer: `1157920892373161954235709850086879078532699846656405640394`
this, the coordinates of the points are 256-bit integers in a big modulo **p**.The secp256k1 curve was rarely used before the advent of Bitcoin. As you can imagine, after it was used for Bitcoin, it became popular. Unlike most of her sisters, which commonly use a *random* structure, secp256k1 has been crafted to be more efficient. Indeed, if the implementation is optimized, this curve is 30% faster than others. Moreover, the constants (**a** and **b**) have been selected by the creator with a lower possibility of injecting a backdoor into it, which is different from the **National Institute of Standards and Technology** (**NIST**)'s other curves.**Please note that this last sentence is valid until proven otherwise.**Finally, in secp256k1, the generator (**G**), the order (**n**), and the prime (modulo **p**) are not randomly chosen but are functions of other parameters. All that makes this curve one of the best you can implement and it's useful for the scope of digital signatures. This is the reason why Bitcoin's

developers chose it. In Bitcoin, precisely, we will see how many digits will be selected for the modulo (**p**), the order (**n**), and the base point (**G**) to make secp256k1 secure.Let's go on now to explore how the digital signature ECDSA algorithm is implemented in secp256k1.

Step 1 – Generating keys

The **[d]** private key is a number randomly chosen within the range **1 ≤ k ≤ n-1**, where (**n**) is the order of **G**. This number has to be of a length of 256 bits so that it computes the hash value of the random number with SHA-256, which gives as output a 256-bit number. This is the way to be sure that the number is effectively 256 bits in length. Remember that if the output gives a number lower than (**n-1**), then the key will be accepted; if not, it will make another attempt.The public key (**KPub**) is derived by the following equation: `KPub = Kpriv*G` Thus, the number of possible private keys is the same as the order of **G**: **n**.To calculate the public key (**Q**), starting from the private key, **[d]**, in secp256k1, we have to use the following equation: `Q ≡ [d] * G (mod p)` The result of the equation will be **Q** (the public key).After this operation, we can switch to calculating the digital signature. Suppose that Alice is the sender of the signature and her **[d]**, (**G**), and (**Q**) parameters have already been calculated. In a real case, Victor is the verifier (the miner) who has to verify the true ownership. Let's go on to see how to sign a Bitcoin transaction in secp256k1.

Step 2 – Performing the digital signature in secp256k1

Remember that to sign a document, **[M]**, or, even better in this case, a value in Bitcoin, **[B]** (as we discussed in *Chapter 4*, *Introducing Hash Functions and Digital Signatures*), it is always recommended to compute **Hash[B] = z**.So, we will sign (**z**), which is the hash of the message, and *not* **[M]** directly. Another parameter that we need is **[k]**, which is an ephemeral key (or session key).The sub-steps to perform the digital signature (**S**) are similar to those for a public/private key algorithm. The difference here is that the discrete logarithm is obtained through a scalar multiplication:

- Alice chooses a random secret, **[k]**, which gives us the following: **[1≤ k ≤ n-1]**.

- She calculates the coordinates, **R(x,y) = k*G**.
- Alice finds **r ≡ x (mod n)** (the *x* coordinate of **R(x,y)**) .
- Alice calculates **S ≡ (z + r*d)/ k (mod p)** (this is the digital signature).

Alice sends the (**r** and **S**) pair to Victor for verification of the digital signature.

Step 3 – Verifying the digital signature

Victor receives the (**r** and **S**) pair. He can now verify whether (**S**) has really been performed by Alice.To verify (**S**), Victor will follow this protocol:

- Check whether (**r**) and (**S**) are both included between **1** and (**n-1**).
- Calculate **w ≡ S^(-1) (mod n)**.
- Calculate **U ≡ z * w (mod n)**.
- Calculate **V ≡ r * w (mod n)**.
- Compute the point in secp256k1: **R (x,y) = UG + VQ**.
- Verify that **r ≡ Rx (mod n)**.

If (**r**) corresponds to **Rx (mod n)** – the *x* coordinate of the **R** point – then the verifier accepts the signature (**S**) corresponding to the value of Bitcoin, **[B]**, the owner of which claims to own the amount of Bitcoin declared.Without the support of a practical example, it is rather difficult to understand such a complex protocol. Some of the operations performed on secp256k1 for the digital signature of a Bitcoin transaction are quite complex to interpret and need a practical example to understand.

A numerical exercise on a digital signature on secp256k1

In this section, we will deep dive into the digital signature of secp256k1 in order to understand the mechanism behind the operations of implementation and validation of the digital signature.Suppose for instance that the parameters of the curve are the following: `p = 67 (modulo p)G = (2,22)Order n = 79Private Key: [d]=` as we have chosen a very simple private key, it is just enough to perform a double point to obtain the public key (**Q**): `Q = d*G` In this case, we proceed to calculate the public key (**Q**). First, we will be using the following formula to calculate the double point: `t = (3XP^2 + a)/ 2YPt = (3*2^2 + 0)/ 2*22= 12/ 44 = Reduce[4` **Q = d * G** using numbers implies replacing **[d]** and (**G**) with numbers: `Q = 2 *(2,22)` Relying on the formula of the double point, let's find the coordinates of **Q** (*x* and *y*), starting with *x*: `xQ = t^2 - 2XGxQ ≡ 49^2 – 2 * 2 = 52 (mod 67)xQ = 52` In the same way, let's find the *y* coordinate of **yQ**: `yQ = t(xG - xQ) - yGxyQ ≡ 49 (2 -52) - 22 = 7 (mod 67)yQ = 7` S the coordinates of the public key (**Q**) are as follows: `Q (x,y) = (52, 7)` Now, Alice can perform her digital signature (**S**). First, Alice computes the **H[M]= z** hash.Suppose, **z = 17** is the hash of the **[B]** value of the Bitcoin.Alice chooses a random secret number for the **[k]** session key: `k = 3` Alice calculates the **R(x,y) = k*G** coordinates: `k*G = 3 * (G) = G + 2G` We have already calculated **2G = (52,7),** so it's possible to rely on the additional formula to calculate **G + 2G = (2,22) + (52,7)**. Let's recall the formulas of point addition on elliptic curves: `t1 = (yQ-yG)/(xQ-xG)` To compute **xR** (the *x* coordinate of the **R** point), we have to solve this operation: `t1 ≡ (7 – 22)/(52 – 2) (mod 67)t1 ≡ 60 (mod 67)` We are seeking **[xR]**: `xR ≡ t1^2 - (xG + xQ) (mod 67)xR ≡ 60^2 - (2+52) = 62 (mod 6` consider the *x* coordinate of **R**: `r = 62 (mod 79) = 62` At this point, we can perform the signature applying the formula: `S ≡ (z + r*d)/k (mod p)S ≡ (17 + 62 * 2)/3 (mod 67) = 47` V have gained a very important point – the signature (**S**): `S = 47` Alice sends the (**S = 47, r = 62**) pair to Victor.To verify the digital signature, Victor receives **(S, r) = (47, 62)**.The public parameters that Victor has available are as follows: `z = 17n = 79 (order of G)G = (2,22) Base PointQ = (52,7) P` of all, Victor has to check the following: `1≤ (47, 62) ≤ (79-1)` In fact, here we go – the signature passes the first check.Victor now calculates (**w**), the inverse of the digital signature (**S**): `w ≡ S^(-1) (mod n)` That, as we have already seen on several occasions, means to perform the inverse functions of **S (mod n)**: `Reduce[(S)*x == 1, x, Modulus -> (n)]= Reduce[(47)*x == 1, x,` Victor calculates (**U**): `U ≡ z * w (mod n)U ≡ 17 * 37 (mod 79)U = 76` Now Victor is able to verify the signature: `V ≡ r * w (mod n)V ≡ 62 * 37 (mod 79)V = 3` The game is not finished yet; we have to *convert* through scalar multiplication the coordinates of **V = 3** on secp256k1.To do that, we have to perform the following equation: `R(x,y) = U*G + V*Q` We will split the preceding equation into two parts: (**UG**) and (**VQ**). We start by calculating **UG**: `U*G = 76G = 2*38G = 2*(2*19G) = 2*(2(G +18G) = 2*(2 (G + 2(9G` order to reduce **76G** through scalar multiplication, we have to perform six-point double (**G**) operations and two-point additions.This shortcut will come in handy when the numbers are very large. There is *no* efficient algorithm able to perform such an operation of reduction like this, so we have to use our brain.We already know the results of **2G = 2 (2,22) = (52,7)**, so we can reformulate the preceding operations, **2*(52,7) = (21, 42)**, already calculated by me, and we arrive at this further reduction: `UG = 2 *((52,7) + 2((2,22) + (21,42))) = 2 *((52,7) + 2(` the next step, we have to calculate **VQ = 3Q = Q+ 2Q.** One method to perform this scalar multiplication is first to calculate **2Q = 2 *(52,7) = (25,17)**.Then we proceed to compute **Q + 2Q = (52,7) + (25,17)**: `VQ = (52,7) + (25,17) = (11,20)VQ = (11,20)` Finally, we can add **UG + VQ** in a unique adding point: `R (x,y) = U*G + V*QR (x,y) = (62,4) + (11,20)R (x,y) = (62,6` the last verification, Victor can check the following: `r ≡ Rx (mod n)` In fact, it turns out as follows: `r = 62 = Rx = 62 (mod 79)` So, finally Victor accepts the signature!We have seen how complex it is even if we use small numbers to delve inside this protocol, as we have demonstrated in this example. So, you can imagine the enormous complexity that must be involved if the parameters are 256 bits or more. This elliptic curve is proposed to protect the ownership of Bitcoin. The signer, through their signature (**S**), can demonstrate they own the **[B]** value corresponding to the computed hash (**z**). It's verified that secp256k1 is a particular elliptic curve, as we saw at the beginning of this section when I mentioned the characteristics of this curve, which, being different from others, is more efficient and has parameters chosen in a particular way to be implemented.If you are curious about the implementation and the digital signature ECDSA algorithm, you can visit the NIST web page and the **Institute of Electrical and Electronics Engineers** (**IEEE**). NIST has published a list of different kinds of ECC and the recommended parameters of the relative implementations. You can find the document at this link: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-78-4.pdf.IEEE is a private organization dedicated to promoting publications, conferences, and standards. This organization released the IEEE P1363-2000 (Standards Specification for Public-Key Cryptography), where it is possible to find the specifications to implement the ECC.In the next section, you will find an attack against EDCSA private keys made by a hacker group calling itself *fail0verflow*, which announced it had recovered the secret keys used by Sony to sign in to the PlayStation 3. However, this attack worked because Sony didn't properly implement ECDSA, using a static private key instead of a random one.

Attacks on EDCSA and the security of elliptic curves

This attack on ECDSA can recover the private key, **[d]**, if the random key (ephemeral key), **[k]**, is not completely random or it is used multiple times for signing the hash of the message (**z**).This attack, implemented to extract the signing key used for the PlayStation 3 gaming console in 2010, recovered the keys of more than 77 million accounts.To better understand this disruptive attack (because it will recover not only the message but also the private key, **[d]**), we will divide it into two steps. In this example, we consider the case when two messages, **[M]** and **[M1]**, are digitally signed using the same private keys, **[k]** and **[d]**.

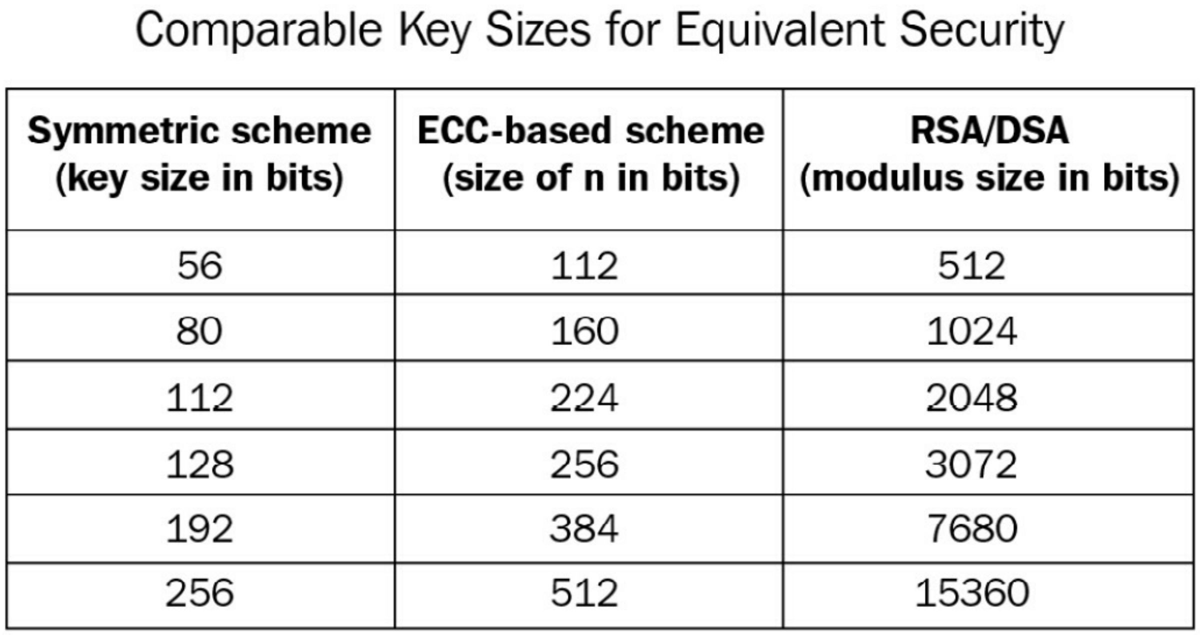
Step 1 – Discovering the random key, [k]

The signature (**S = 47**) generated at the time (**t0**) from the hash of the message, **[M]**, as we know, is given by the following mathematical passages: `S ≡ (z + r*d )/k (mod p)` Here it is presented in numbers: `S ≡ (17 + 62 * 2)/3 (mod 67) = 47S = 47` Suppose now we know **z1 = 23** (the hash of the second message, **[M1]**) transmitted at the time (**t1**) and generating the signature (**S1**). Moreover, we are supposed to know the second signature (**S1**) given by the equation: `S1 ≡ (z1 + r*d)/k (mod p)` Using the **Reduce** function of

Mathematica, we have the following result for (**S1**):
`[k*x = (z1 + r*d), x, Modulus -> (n)] = 49 S1 = 49` Given **(S – S1)/(z-z1) ≡ k (mod n)**.Substituting the parameters in the preceding equation with the numbers of our example, we can easily gain **[k]** using the **Reduce** modular function of Mathematica, as follows: `Reduce[(47 - 49)*x == (17 - 23), x, Modulus -> (79)]k=3` Af we get **[k]**, we can also gain the private key, **[d]**. Let's see what happens in the next step.

Step 2 – Recovering the private key, [d]

After we have recovered the random key, **[k]**, we can easily compute the private key, **[d]**. We know that (**S**) is given by the following equation: `S = (z + r*d )/k` Switching (**k**) from denominator to enumerator at the left side, we can write the equation in the following form: `S*k = z + r*d` From knowing (**k**) = 3, we can find **[d]** because also all the other parameters (**S**,**z**,**r**) are public.So, we can find **[d]** by writing the previous equation as follows: `d = (S*k –z)/r` All the parameters on the right side are known.We can write it out in numbers, as follows: `(47 * 3 – 17)/r = 2` That is the number of the private key, **[d]= 2**, discovered!Analyzing this attack, we come to a question: what happens if someone can find a method to generate multiple signatures (**S1**, **S2**, and **Sn**) starting from (**z1**, **z2**, and **zn**) or generate multiple signatures starting from (**z**)?In other words, as (**z**) comes from the hash of the message (the value of Bitcoin recovered in a wallet) that we have called **[B]**, what happens if we can generate an **S1** signature starting from (**z**)?I invite you to think about this question because if that is possible, then it should be possible to generate multiple signatures on a single transaction, all verified by the receivers. Let's now see a brief analysis of ECC's computational power compared to other encryptions. Elliptic curve efficiency compared with the classic public/private key cryptography algorithm is shown in the following table. You can immediately perceive the lowest key size used in elliptic curves compared to the RSA/DSA algorithms. The equivalence you see in the following figure also takes into consideration the symmetric scheme key size (**Advanced Encryption Standard** (**AES**), for example). This scheme is undoubtedly shorter than the elliptic curve, but here is a comparison because, as you already know, the elliptic curve is able to spread out digital signatures differently by symmetric scheme encryption, so the effective comparison has to be done with asymmetric encryption:



Comparable Key Sizes for Equivalent Security

Symmetric scheme (key size in bits)	ECC-based scheme (size of n in bits)	RSA/DSA (modulus size in bits)
56	112	512
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

Figure 7.9 – William Stallings' table of comparison – ECC versus classical cryptography

As you can see, 256-bit encryption performed on ECC is equivalent to a 3,072-bit key on RSA. Moreover, we can compare a 512-bit key on elliptic curve to a key size on RSA of 15,360 bits. Compared with symmetric encryption (AES, for example), ECC needs double the amount of bits.

Important Note

A warning regarding the key's length: it doesn't matter how long the modulus is or how big the key size is – if an algorithm logically breaks out, nothing will repair its defeat.

Considerations about the future of ECC

Now that we have seen how a practical attack on ECDSA works, one of the most interesting questions we should ask for the future is the following: *Is elliptic curve cryptography resistant to classical and quantum attacks?*At a glance, the answer could be that most elliptic curves are not vulnerable (if well implemented) to most traditional attacks, except for the same ones we find against the classic discrete logarithm (such as Pollard Rho or a birthday attack) and man-in-the-middle attacks in D–H ECC. In the quantum case, however, Shor's algorithm can probably solve the elliptic curve problem, as we will see in the next chapter, dedicated to quantum cryptography.Thus, if someone asks: are my Bitcoins secured for the next 10 or 20 years? We can answer: under determinate conditions, yes, but if the beginning of the quantum-computing era generates enough qubits to break the classical discrete logarithm problem, it will probably break the ECC discrete logarithm problem in polynomial time too. So, I agree with Jeremy Wohlwend (a Ph.D. candidate at MIT), who wrote about *Elliptic Curve Cryptography: Pre and Post Quantum*:*"Sadly, the day that quantum computers can work with a practical number of qubits will mark the end of ECC as we know it."*

Summary

In this chapter, we have analyzed some of the most used elliptic curves. We have seen what an elliptic curve is and how it is designed to be used in cryptography. ECC has algorithms and protocols designed mainly to cover secrets related to public/private encryption systems, such as the D–H key exchange and the digital signature. In particular, we analyzed the discrete logarithm problem transposed into ECC, so we have familiarized ourselves with the operations at the core of ECC, such as adding points on the curve and scalar multiplications.These kinds of operations are quite different from the addition and multiplication we are familiar with; here, indeed, lies the strength of elliptic curves.After the experimentation done on D–H ECC, we analyzed in detail secp256k1, which is the elliptic curve used to implement digital signatures on Bitcoin protocol through the ECDSA.So, now that you have learned about elliptic curves and systems as alternative methods in public/private encryption, you can understand that one of the best properties of these curves is the grade of efficiency in their implementation, which can be a lower key size.At the end of this chapter, we asked a question about ECC's robustness regarding quantum attacks. It was with this answer that I introduced the topic of *Chapter 8*, *Quantum Cryptography*. This chapter will, for sure, cover one of the most intriguing and bizarre topics of this book. Quantum computing and quantum cryptography will be the new challenge for cryptography's future.