EXPERT INSIGHT

Cryptography Algorithms

Explore New Algorithms in Zero-knowledge, Homomorphic Encryption, and Quantum Cryptography

Second Edition

Massimo Bertaccini, PhD

packt

Cryptography Algorithms

Second Edition

Explore New Algorithms in Zero-knowledge, Homomorphic Encryption, and Quantum Cryptography

Massimo Bertaccini, PhD

Cryptography Algorithms

Second Edition

Senior Publishing Product Manager: Reshma Raman

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Namrata Katare

Content Development Editor: Matthew Davies

Copy Editor: Safis Editing

Technical Editor: Aneri Patel

Proofreader: Safis Editing

Indexer: Manju Arasan

Presentation Designer: Pranit Padwal

Developer Relations Marketing Executive: Meghal Patel

First published: Feb 2022

Second edition: August 2024

Production reference: 1070824

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83508-003-0

`www.packt.com`

To my mom and my dad

- Massimo

Contributors

About the author

Massimo Bertaccini, **PhD**, is a researcher, principal scientist, CEO, and co-founder at Cryptolab Inc. He holds several patents in cryptography, quantum cryptography, and AI. His career started as a professor of mathematics and statistics, following which he founded Cryptolab Inc., a start-up in the field of cryptography solutions for cybersecurity. With his team of engineers, he designed and implemented the first search engine in the world that can work with encrypted data.

He has obtained several international prizes and awards, such as the Silicon Valley Inventors award, the Seal of Excellence from the EU, and Security Solutions Provider of the Year – USA, 2023. Currently, as a contract professor, he teaches cryptography for a cybersecurity course and has published many articles in the field of cryptography and blockchain.

The first edition of *Cryptography Algorithms* was the tenth bestseller in its category on Amazon for 40 weeks and proclaimed by BookAuthority as the best book of 2023 in homomorphic and quantum encryption.

Special thanks to Marco Massari and Daniele Sartini, and a particular special thanks with all my heart to Leon Xu and the other reviewers.

I would also like to thank Paul Heger, Matteo Sarica, Davide Carmeci, Maicol Lombardi, Jack Wolosewicz, Piergiorgio Montanari, Gaudenzio Garavini, Yijing Gu and Sara Rauchwerger for their support during my professional career.

Finally, I would like to thank all of the Packt staff who helped me during the writing of this book.

About the reviewers

David Tillemans has more than 20 years of experience in security and secure development. He worked for 10 years with cryptography and smart cards during the development of a public key infrastructure product. He later switched from being a cryptography engineer to an application security engineer. In this function, he implemented a secure development process, pivoting from cryptographer to ethical hacker and adviser on secure development processes.

Because of his background, he started to specialize in the combination of cryptography, ethical hacking, and secure development practices concerning cryptography. Today, he is an independent consultant in the financial sector and government sector, where he acts as PKI architect, a secure developer, and a business security consultant.

Dr. Paul Duplys is a security researcher and lead of the Security, privacy & safety research program in the Corporate Research division of Robert Bosch GmbH, the world's largest tier-1 automotive supplier and manufacturer of industrial, residential, and consumer goods. Paul has been doing applied research in various fields of information security since 2007. His current research interests include security automation, software security, network security, intrusion detection and honeypots, AI applications for security and security of AI, privacy engineering, and privacy preserving technologies. Paul holds a PhD degree in computer science from the University of Tuebingen.

Leon Xu is a software engineer at TVU Networks, a leading company in innovative media supply technology. In his role, he focuses on optimizing network transmission algorithms and developing real-time object detection and recognition applications.

He holds an M.S. degree in applied physics from Stanford University. He is a passionate enthusiast of quantum computing, for which he engages in events such as the IBM Quantum Challenge and QHack.

> *I am honored to be a reviewer of this book and bring my technical expertise to the success of this publication.*

About the beta readers

The following list comprises readers from our beta program who kindly guided the development of this edition through their feedback. We would like to thank the following individuals for their valuable assistance reviewing this edition:

- Thomas Morris – AgileDinosaur
- Anjali Lathiya – Archangel
- Aryan Pandey – oumuamua

Table of Contents

Preface xvii

Section 1: A Brief History and Outline of Cryptography 1

Chapter 1: Deep Dive into Cryptography 3

An introduction to cryptography 4

Binary numbers, ASCII code, and notations • 7

Fermat's Last Theorem, prime numbers, and modular mathematics • 9

A brief history and a panoramic overview of cryptographic algorithms 14

Rosetta Stone • 14

The Caesar cipher • 16

ROT13 • 19

The Beale ciphers • 20

The Vernam cipher • 26

Notes on security and computation 29

Summary 32

Section 2: Classical Cryptography (Symmetric and Asymmetric Encryption) 35

Chapter 2: Symmetric Encryption Algorithms 37

Notations and operations in Boolean logic 38

DES algorithms **43**

Simple DES • 44

Bit initialization • 46

Bit expansion • 46

Key generation • 48

Bit encryption • 49

DES • 50

Key generation in DES • 51

Analysis of the DES algorithm • 57

Violation of DES • 60

Triple DES • 61

DESX • 63

AES Rijndael **64**

Description of AES • 64

Attacks and vulnerabilities in AES • 69

Summary **74**

Chapter 3: Asymmetric Encryption Algorithms 77

Introduction to asymmetric encryption **77**

The pioneers • 79

The Diffie-Hellman algorithm **80**

The discrete logarithm • 82

Explaining the D-H algorithm • 84

Analyzing the algorithm • 85

Possible attacks and cryptanalysis on the D-H algorithm • 86

RSA **88**

Explaining RSA • 90

Analyzing RSA • 91

Conventional attacks on the algorithm • 92

The application of RSA to verify international treaties • 94

Unconventional attacks • 96

PGP 99

The ElGamal algorithm 100

Summary 104

Chapter 4: Hash Functions and Digital Signatures 105

A basic explanation of hash functions 106

Overview of the main hash algorithms 109

Logic and notations to implement hash functions • 110

Explanation of the SHA-1 algorithm • 115

Notes and example on SHA-1 • 119

Example of one block encoded with SHA- 1 • 121

Authentication and digital signatures 124

RSA digital signatures • 125

Why do digital signatures work? • 129

Digital signatures with the ElGamal algorithm • 130

Blind signatures • 133

Blind signature with RSA • 134

Notes on the blind signature protocol • 136

Summary 138

Section 3: New Cryptography Algorithms and Protocols 139

Chapter 5: Zero-Knowledge Protocols 141

The main scenario of a ZKP – the digital cave 142

Non-interactive ZKPs 144

Demonstration of a non-interactive ZKP • 148

Demonstrating an attack on an RSA ZKP • 148

Schnorr's interactive ZKP 151

A demonstration of an interactive ZKP • 152

A challenge for a disruptive attack on an interactive ZKP • 155

One-round ZKP **156**

How it works mathematically • 157

Numerical example • 157

Notes on the one-round protocol • 159

An introduction to zk-SNARKs – spooky moon math **160**

Understanding how a zk-SNARK works • 162

Demonstrating an attack on a zk-SNARK protocol • 163

ZK13 – a ZKP for authentication and key exchange **166**

ZK13 explained • 168

Demonstrating the ZK13 protocol • 170

Notes and possible attacks on the ZK13 protocol • 171

Possible attacks on ZK13 • 171

Summary **171**

Chapter 6: New Inventions in Cryptography and Logical Attacks **173**

The genesis of the MB09 algorithm and blockchain **174**

Introducing the MB09 algorithm and an attempt at demonstrating Fermat's Last Theorem **179**

An extensive explanation of the MB09 algorithm • 183

Introducing the MBXI algorithm **191**

A numerical example of MBXI • 194

Notes on the MBXI algorithm and the prelude to an attack on RSA • 196

Unconventional attacks and self-reverse decryption on RSA **198**

A new protocol to protect RSA and asymmetric algorithms from spying **205**

Digital signatures on MBXI **206**

A direct signature method in MBXI • 208

The appendix signature method with MBXI • 210

A mathematical demonstration of the MBXI digital signature algorithm • 211

The evolution of MB09 and MBXI: an introduction to MBXX **214**

An explanation of the MBXX protocol • 217

Notes on the MBXX protocol • 221

Conclusions on the MBXX protocol • 223

Lightweight encryption **224**

Encryption with Cybpher • 227

Summary **231**

Chapter 7: Elliptic Curves 233

An overview of elliptic curves **234**

Operations on elliptic curves **234**

Scalar multiplication • 239

Implementing the D-H algorithm on elliptic curves **242**

Elliptic curve secp256k1: the Bitcoin digital signature **248**

Step 1: Generating keys • 251

Step 2: Performing the digital signature in secp256k1 • 251

Step 3: Verifying the digital signature • 252

A numerical exercise on a digital signature on secp256k1 .... **252**

Attacks on ECDSA and the security of elliptic curves **258**

Step 1: Discovering the random key, [k] • 258

Step 2: Recovering the private key, [d] • 259

Considerations about the future of ECC **261**

Summary **261**

Chapter 8: Homomorphic Encryption and Crypto Search Engine 263

Introduction to CSE: homomorphism **264**

Partial homomorphism in RSA **266**

Analysis of homomorphic encryption and its implications **270**

Math and logic behind search engines • 271

Introduction to trees: graph theory **276**

Huffman code **278**

Hash and Boolean logic **281**

CSE explained **282**

The innovation in CSE **284**

Computational analysis on CSE 288

Example of computational brute-force cracking • 290

Applications of CSE 292

The new frontier of CSE and a new quantum algorithm for message transmission: QTM 294

Summary 295

Section 4: Quantum Cryptography 297

Chapter 9: Quantum Cryptography 299

Introduction to Q-Mechanics and Q-Cryptography 300

An experiment that changed the story of quantum • 301

An imaginary experiment to understand the elements of Q-Mechanics 302

Step 1: superposition • 302

Step 2: the indeterminacy principle • 304

Step 3: spin and entanglement • 306

Origin of Q-Cryptography: quantum money 310

QKD: BB84 313

Step 1: initializing the quantum channel • 315

Step 2: transmitting the photons • 315

Step 3: determining the shared key • 316

Analysis of attack and technical issues 318

Quantum computing 323

Shor's algorithm 326

Hypothesis and thesis • 326

Step 1: initializing the qubits • 327

Step 2: choosing the random number, (a) • 328

Step 3: quantum measurement • 329

Step 4: finding the right candidate, (r) • 330

QFT • 330

Step 5: factorizing (n) • 335

Notes on Shor's algorithm • 335

Post-Q-Cryptography 336

Summary 337

Chapter 10: Quantum Search Algorithms and Quantum Computing 339

An overview of Grover's algorithm 340

Elements of quantum programming: quantum information and circuits 342

Classical information • 343

Quantum information, gates, and circuits • 346

Pauli gates (X, Z) • 347

Identity gate • 350

Hadamard gate • 351

Deep dive into Grover's algorithm 353

Pseudocode for running Grover's algorithm • 355

Step 1: Allocate the register of qubits in superposition • 356

Step 2: Iterations on Grover's operator: G • 360

The unique search problem and the amplitude amplification probability • 362

Step 1: The promise of the inputs • 363

Step 2: Grover's operation G • 364

Step 3: Quantum measurement • 369

Summary 370

Other Books You May Enjoy 375

Index 379

Preface

In this age of high connectivity, cloud computing, ransomware, and hackers, digital assets are changing the way we live our lives. And so, cryptography and cybersecurity are crucial. The changes in processing and storing data have required adequate evolution in cryptographic algorithms to advance in the eternal battle against information piracy.

Starting from the basics of symmetric and asymmetric algorithms, I will describe the modern techniques of authentication, transmission, and searching used on encrypted data to shelter it from spies and hackers. You'll encounter algorithms with zero-knowledge protocols, elliptic curves, homomorphic search, and quantum cryptography.

The possibilities of quantum cryptography continue to grow and will break measures previously thought to be secure. Innovations in quantum cryptography are at the forefront of cryptographic thought, so in this second edition, I have endeavored to give you a primer in quantum cryptography and an early introduction to quantum search, looking at the Grover algorithm.

I see this book as a tool for students and professionals who want to focus on the next generation of cryptography algorithms. I want to help you be aware of the modern developments in cryptography, but first, my focus is on teaching the mathematical logic of these algorithms to help you understand the fundamentals. As you go through the book, I'm hoping you'll gradually find what areas of practical implementation would interest you most in your career.

Who this book is for

This book is for students, IT professionals, cybersecurity enthusiasts, or anyone who wants to develop their skills in modern cryptography and build a successful cybersecurity career.

What this book covers

Section 1: A Brief History and Outline of Cryptography

Chapter 1, *Deep Dive into Cryptography*, introduces cryptography, what it is needed for, and why it is so important in IT. This chapter also provides a panoramic view of the principal algorithms in the history of cryptography.

Section 2: Classical Cryptography (Symmetric and Asymmetric Encryption)

Chapter 2, *Symmetric Encryption Algorithms*, analyzes symmetric encryption. We will focus on algorithms such as DES, AES and Boolean logic, which are widely used to implement cybersystems. Finally, we will showcase attacks to these algorithms.

Chapter 3, *Asymmetric Encryption Algorithms*, analyzes the classical asymmetric encryption algorithms, such as RSA and Diffie–Hellman, and the main algorithms in private/public key encryption.

Chapter 4, *Hash Functions and Digital Signatures*, focuses on hash functions such as SHA-1 and looks at digital signatures, which are one of the pillars of modern cryptography. We will look at the most important and famous signatures and, as a particular case of anonymous signatures, blind signatures.

Section 3: New Cryptography Algorithms and Protocols

Chapter 5, *Zero-Knowledge Protocols*, looks at zero-knowledge protocols, which are one of the new fundamental encryption protocols for blockchain. They are very useful for authenticating humans and machines without exposing any sensitive data in an unsafe channel of communication. New protocols, such as zk-SNARK, used in the blockchain are based on these algorithms. Finally, we will present Z/K13, which is a new protocol I invented in zero knowledge.

Chapter 6, *New Inventions in Cryptography and Logical Attacks*, presents three algorithms I invented. MB09 is based on Fermat's Last Theorem. MB11 could be an alternative to RSA. Digital signatures related to these algorithms are also presented. Moreover, we will present MBXX, a new protocol, which can be used for consensus.

Chapter 7, *Elliptic Curves*, looks at the new frontier for decentralized finance, elliptic curves. Satoshi Nakamoto adopted a particular kind of elliptic curve to implement the transmission of digital currency in Bitcoin, called SECP256K1. We'll see how it works and what the main characteristics of this very robust encryption are.

Chapter 8, *Homomorphic Encryption and Crypto Search Engine*, looks at the crypto search engine, which is an application of homomorphic encryption. It is a search engine able to search for a query inside encrypted content. We will see how it has been implemented, the story of this enterprise, and the possible applications of this disruptive engine for security and data privacy.

Section 4: Quantum Cryptography

Chapter 9, *Quantum Cryptography*, looks at how, with the advent of quantum computing, most of the algorithms we have explored until now will be under serious threat of brute-force attacks. One possible solution is quantum cryptography. It is one of the most exhilarating and fantastic kinds of encryption that the human mind has invented. We are only at the beginning of quantum cryptography, but it will be widely adopted in a short time.

Chapter 10, *Quantum Search Algorithms and Quantum Computing*, introduces Grover's algorithm as an example of quantum search and attacks to classical symmetric encryption. By learning the logic of this algorithm, we will see how some elements of quantum computing are applied in cryptography, in particular for problems related to random searching and brute force attacks.

To get the most out of this book

This book systematically addresses mathematical issues related to the algorithms that may arise. However, a prior knowledge of university-level mathematics, algebra, its main operators, modular mathematics, and finite fields theory is required. Some knowledge of elliptic curves and quantum computing, especially matrices and plotting curves, would also be beneficial to get the most out of this book.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781835080030`.

Conventions used

There are a number of text conventions used throughout this book.

Italics is used to indicate mathematical equations and algebraic characters within text, as well as the typical use of placing emphasis. For example, "First, we calculate $2^4=16$.

Mathematical equations are set as follows:

$$a^n + b^n = z^n$$

Bold: Indicates a new term, an important word, or words that you see onscreen. Here is an example: "A **cipher** is a system of any type able to transform plaintext (a message) into not-intelligible text (a ciphertext or cryptogram)"

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share your thoughts

Once you've read *Cryptography Algorithms, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1

A Brief History and Outline of Cryptography

This section is an introductory part and aims to provide basic definitions, information, and a history of cryptography and its algorithms.

This section comprises the following chapter:

- *Chapter 1, Deep Dive into Cryptography*

1

Deep Dive into Cryptography

Welcome to the world of cryptography. In this book, you will discover the secrets behind this fascinating science that may be very important for your career as well as for your general knowledge. At the end of this book, you will be able to understand the most important algorithms that make cryptography great and have discovered some of the new ones invented and implemented by me in my career. I hope your journey in reading this book will be enjoyable, and that you will reach your academic and professional goals.

This chapter introduces you 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 cipher to the Vernam cipher and other lesser-known algorithms, such as the Beale ciphers. Then, **Rivest-Shamir-Adleman** (**RSA**), Diffie–Hellman, **Advanced Encryption Standard** (**AES**), zero-knowledge, elliptic curves, homomorphic encryption, quantum cryptography, and other famous algorithms will be described in detail in the proceeding part of this book. Finally, this chapter will help you understand 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 **American Standard Code for Information** (**ASCII**) code
- Fermat's Last Theorem, prime numbers, and modular mathematics
- The history of the principal cryptographic algorithms and an explanation of some of them (Rosetta, Caesar, ROT13, Beale, and Vernam)

- Security notation (semantic, provable, **one-time pad** (**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'll start with a definition of an algorithm.

In mathematics and computer science, an **algorithm** is a finite sequence of well-defined computer-implementable instructions.

Here is an important question: 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):

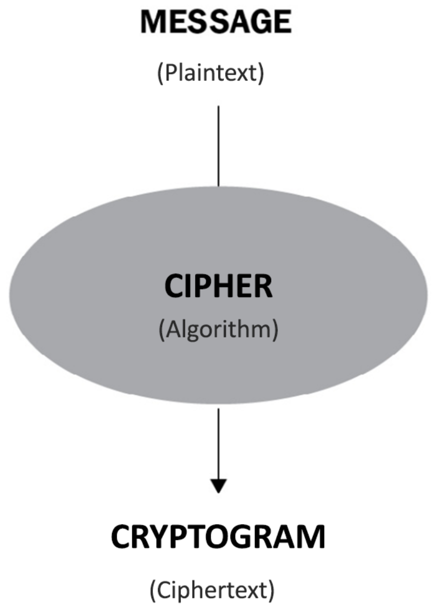


Figure 1.1: Encryption process

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.

In this book, I will analyze many kinds of cryptographic techniques but, essentially, we can group all the algorithms into two big families: symmetric and asymmetric encryption.

We need some more definitions to understand cryptography:

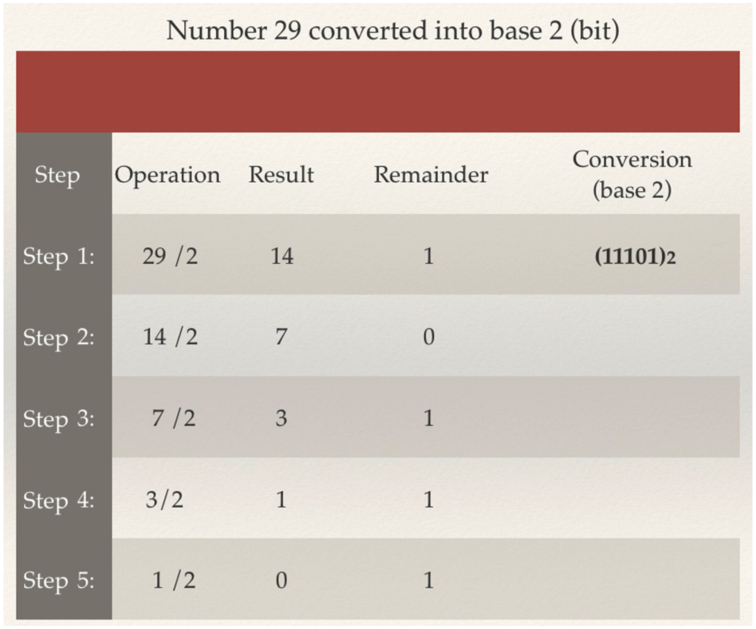
- **Plaintext**: In cryptography, this indicates unencrypted text or everything that could be exposed in public. For example, *(meet you tomorrow at 10 am)* is plaintext.
- **Ciphertext**: In cryptography, this indicates the result of the text after having performed the encryption procedure. For example,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

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)

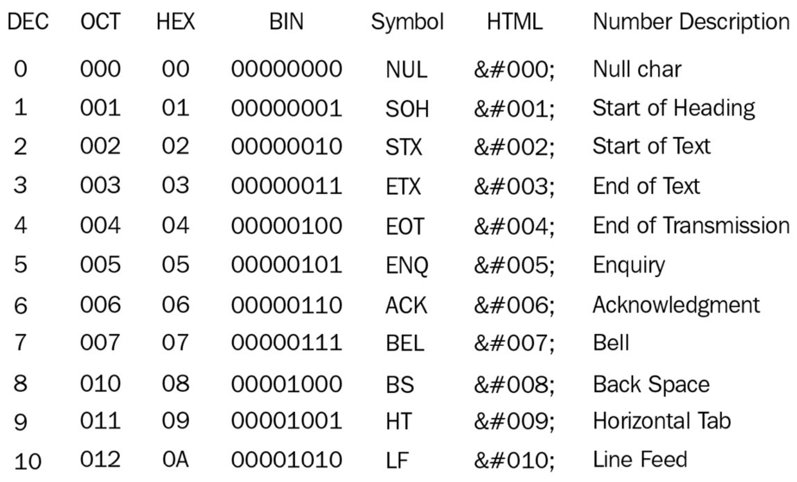
The remainder of a division is very popular in cryptography because **modular mathematics** is based on the concept of remainders. We will go deeper into this topic in the next section, when I explain prime numbers and modular mathematics.

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

In *Chapter 4*, *Hash Functions and Digital Signatures*, we will learn about hex and octal. The key detail at this moment is to observe the binary system.

Note that, in my implementations, made with the **Wolfram Mathematica** research software, I will often use 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:

$$88 \quad 130 \quad 58 \quad 01011000 \quad X \quad \&\#88; \quad Uppercase\ X$$

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 =), are 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 uniformity.

Another symbol, ≈ (approximate), can be used to denote similar approximative elements.

When needed, you will also encounter the ^ (exponent) symbol as a classical way to express exponentiation: *a^x* (*a* elevated to *x*).

The ≠ symbol, as you should remember from high school, means **not equal** or **unequal**, which is represented in modular mathematics 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 a new notation.

Finally, as you should know, 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. You will see that using a composite number instead of a prime number in cryptography could cause a big lack of security or an attack (see *Chapter 5*, *Zero-Knowledge Protocols*).

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, which are foundational to cryptography. 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,* there is no integer, *a*, *b*, or *c*, that verifies the sum:

$$a^n + b^n = c^n$$

Why is this theorem so important to us? This is something that you will come to appreciate over the course of this book as we encounter more algorithms. Essentially, 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:

$$a^p + b^p \neq c^p$$

Here, p is any prime number greater than 2.

Fermat himself noted in a paper that he had a beautiful demonstration of the theorem *that was too large to fit in the margin of his notes,* but it has never been found.

Wiles' proof 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. You will discover part of the logic used in Wiles' proof when you 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 Inventions in Cryptography and Logical Attacks*, 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 where n is some positive integer:

$$2^{2n} + 1$$

If we substitute n with integers, we can obtain some prime numbers:

- $n = 1, p = 5$
- $n = 2, p = 17$
- $n = 3, p = 65$ (not prime)
- $n = 4, p = 257$

Probably more famous but very similar is the Mersenne prime numbers formula, again where n is some positive integer:

$$2^n - 1$$

This yields the following results:

- $n = 1, p=1$
- $n = 2, p=3$
- $n = 3, p=7$
- $n = 4, p=15$ (not prime)
- $n = 5, p=31$

Despite countless attempts to find a formula that exclusively represents all prime numbers, nobody has reached this goal as of yet.

The **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.*
- *The 51st Mersenne prime has been found!*
- *December 21, 2018 — The* **Great Internet Mersenne Prime Search** (**GIMPS**) *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 *2x*, 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 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:

$$13 \equiv 1 \ (mod\ 12)$$

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*:

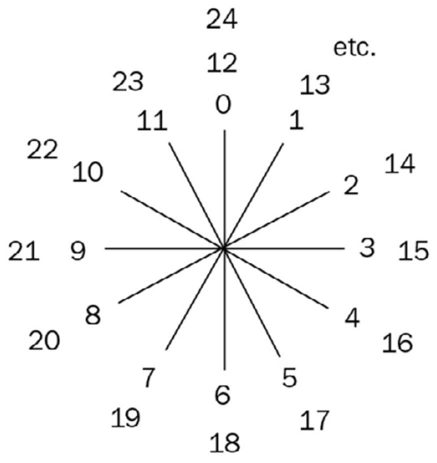


Figure 1.4: Example of modular arithmetics with a clock

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*:

$$a^p \equiv a \ (mod\ p)$$

For example, if *a = 2* and *p = 3*, then *23 = 2 (mod 3)*. In other terms, we find the rest of the division *8 : 3 = 2* with remainder *2*.

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 Little Theorem states that a number, p, is *probably prime* in the following instance:

$$a^p \equiv a \ (mod\ p)$$

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 the transposition of letters in a mechanical fashion, and 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 about 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.

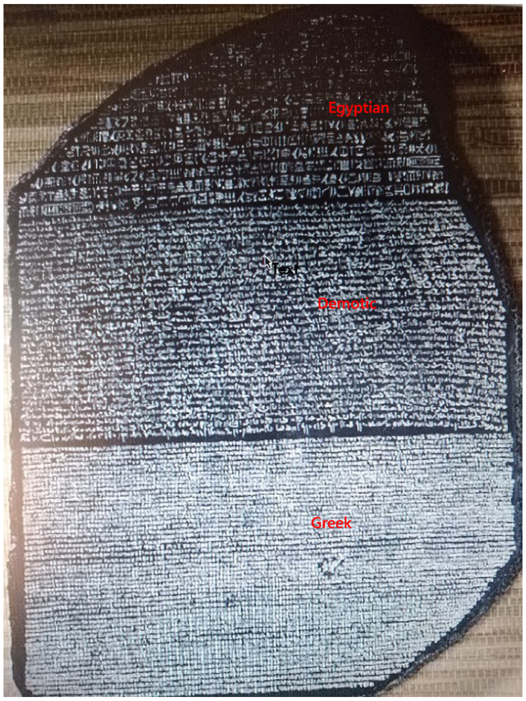


Figure 1.5: The Rosetta Stone with the three languages detected

The Rosetta Stone could only be decrypted because Ancient Greek was well known at the time.

Hieroglyphics were a form of communication between the people of ancient Egypt (and some countries in the surrounding area). Jean-François Champollion has been recognized as the man who deciphered the 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 Thomas Young again in *Chapter 9*, *Quantum Cryptography*. Young was the first to discover the effects of the dualism between waves and particles related to 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. A project called the **Search for Extraterrestrial Intelligence (SETI) Institute** (`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 we get in contact with alien creatures one day, 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.

The 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 soldiers. In fact, what we find is 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 Julius 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]*:

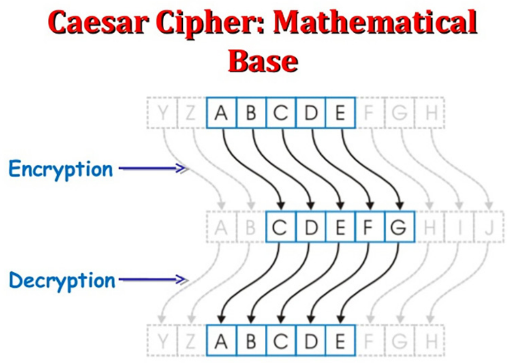


Figure 1.6: The transposition of the letters in the Caesar cipher during the encryption and decryption processes

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 cipher works using *key = +3*; as you'll notice, the word *HELLO* becomes *KHOOR* after encryption, and then it returns to *HELLO* after decryption:

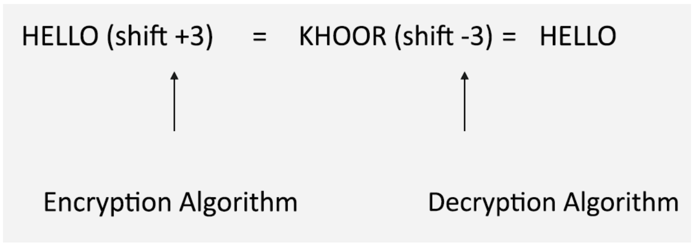


Figure 1.7: Encryption and decryption using the Caesar cipher

As you can imagine, the Caesar cipher 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 use that key to substitute the letters and generate the ciphertext, then things become very interesting. That will be a new kind of cipher called a transposition cipher.

Let's see what happens if we encrypt *HELLO* using a method like this:

1. Write out the alphabet.
2. 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.
3. 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).
4. 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*.

Important note

The following is just an example to show you how the encryption done with a passphrase works. There are obviously many ways of decrypting it, but here we are only considering the encryption.

Let's encrypt *HELLO* with the keyphrase *[JULIUSCAESARJULIUS...]*:

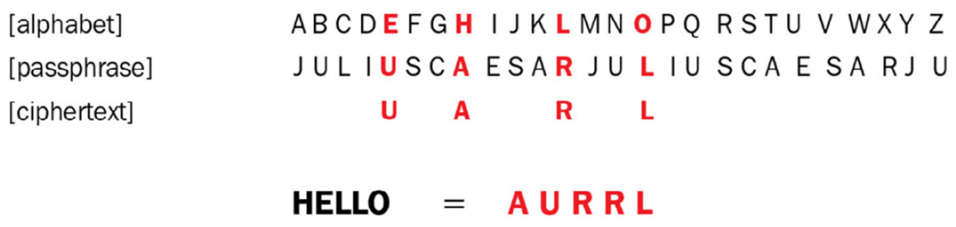

[alphabet] A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[passphrase] J U L I U S C A E S A R J U L I U S C A E S A R J U
[ciphertext] U A R L
HELLO = A U R R L

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 corresponding 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. This means that you multiply *1*2*3...*26*, which results in *403,291,461,126,605,635,584,000,000*.

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,** where ROT is short for "rotate." Essentially, this is a simple cipher derived from the Caesar cipher with a shift of *(+13) using the same ROT13 function to encrypt and decrypt*. Computationally, it is easy to break, just like the Caesar cipher, but it yields an interesting effect: if we shift to the left or 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*. In encryption, ROT13 takes 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:

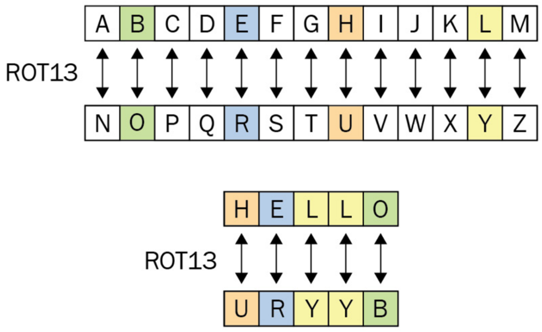


Figure 1.9: The encryption scheme in ROT13

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 ciphers

Going back to the history of cryptography, I would like to show you an amazing method of encryption where the cipher has not yet been decrypted, 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** ciphers.

In order to better understand the method of encryption adopted for these ciphers, 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 do Beale's ciphers look like?

The following content consists of three pages, containing only numbers, in apparently random orders.

The first paper is as follows:

71, 194, 38, 1701, 89, 76, 11, 83, 1629, 48, 94, 63, 132, 16, 111, 95, 84, 341, 975, 14, 40, 64, 27, 81, 139, 213, 63, 90, 1120, 8, 15, 3, 126, 2018, 40, 74, 758, 485, 604, 230, 436, 664, 582, 150, 251, 284, 308, 231, 124, 211, 486, 225, 401, 370, 11, 101, 305, 139, 189, 17, 33, 88, 208, 193, 145, 1, 94, 73, 416, 918, 263, 28, 500, 538, 356, 117, 136, 219, 27, 176, 130, 10, 460, 25, 485, 18, 436, 65, 84, 200, 283, 118, 320, 138, 36, 416, 280, 15, 71, 224, 961, 44, 16, 401, 39, 88, 61, 304, 12, 21, 24, 283, 134, 92, 63, 246, 486, 682, 7, 219, 184, 360, 780, 18, 64, 463, 474, 131, 160, 79, 73, 440, 95, 18, 64, 581, 34, 69, 128, 367, 460, 17, 81, 12, 103, 820, 62, 116, 97, 103, 862, 70, 60, 1317, 471, 540, 208, 121, 890, 346, 36, 150, 59, 568, 614, 13, 120, 63, 219, 812, 2160, 1780, 99, 35, 18, 21, 136, 872, 15, 28, 170, 88, 4, 30, 44, 112, 18, 147, 436, 195, 320, 37, 122, 113, 6, 140, 8, 120, 305, 42, 58, 461, 44, 106, 301, 13, 408, 680, 93, 86, 116, 530, 82, 568, 9, 102, 38, 416, 89, 71, 216, 728, 965, 818, 2, 38, 121, 195, 14, 326, 148, 234, 18, 55, 131, 234, 361, 824, 5, 81, 623, 48, 961, 19, 26, 33, 10, 1101, 365, 92, 88, 181, 275, 346, 201, 206, 86, 36, 219, 324, 829, 840, 64, 326, 19, 48, 122, 85, 216, 284, 919, 861, 326, 985, 233, 64, 68, 232, 431, 960, 50, 29, 81, 216, 321, 603, 14, 612, 81, 360, 36, 51, 62, 194, 78, 60, 200, 314, 676, 112, 4, 28, 18, 61, 136, 247, 819, 921, 1060, 464, 895, 10, 6, 66, 119, 38, 41, 49, 602, 423, 962, 302, 294, 875, 78, 14, 23, 111, 109, 62, 31, 501, 823, 216, 280, 34, 24, 150, 1000, 162, 286, 19, 21, 17, 340, 19, 242, 31, 86, 234, 140, 607, 115, 33, 191, 67, 104, 86, 52, 88, 16, 80, 121, 67, 95, 122, 216, 548, 96, 11, 201, 77, 364, 218, 65, 667, 890, 236, 154, 211, 10, 98, 34, 119, 56, 216, 119, 71, 218, 1164, 1496, 1817, 51, 39, 210, 36, 3, 19, 540, 232, 22, 141, 617, 84, 290, 80, 46, 207, 411, 150, 29, 38, 46, 172, 85, 194, 39, 261, 543, 897, 624, 18, 212, 416, 127, 931, 19, 4, 63, 96, 12, 101, 418, 16, 140, 230, 460, 538, 19, 27, 88, 612, 1431, 90, 716, 275, 74, 83, 11, 426, 89, 72, 84, 1300, 1706, 814, 221, 132, 40, 102, 34, 868, 975, 1101, 84, 16, 79, 23, 16, 81, 122, 324, 403, 912, 227, 936, 447, 55, 86, 34, 43, 212, 107, 96, 314, 264, 1065, 323, 428, 601, 203, 124, 95, 216, 814, 2906, 654, 820, 2, 301, 112, 176, 213, 71, 87, 96, 202, 35, 10, 2, 41, 17, 84, 221, 736, 820, 214, 11, 60, 760

The second paper (which was decrypted) is as follows:

115, 73, 24, 807, 37, 52, 49, 17, 31, 62, 647, 22, 7, 15, 140, 47, 29, 107, 79, 84, 56, 239, 10, 26, 811, 5, 196, 308, 85, 52, 160, 136, 59, 211, 36, 9, 46, 316, 554, 122, 106, 95, 53, 58, 2, 42, 7, 35, 122, 53, 31, 82, 77, 250, 196, 56, 96, 118, 71, 140, 287, 28, 353, 37, 1005, 65, 147, 807, 24, 3, 8, 12, 47, 43, 59, 807, 45, 316, 101, 41, 78, 154, 1005, 122, 138, 191, 16, 77, 49, 102, 57, 72, 34, 73, 85, 35, 371, 59, 196, 81, 92, 191, 106, 273, 60, 394, 620, 270, 220, 106, 388, 287, 63, 3, 6, 191, 122, 43, 234, 400, 106, 290, 314, 47, 48, 81, 96, 26, 115, 92, 158, 191, 110, 77, 85, 197, 46, 10, 113, 140, 353, 48, 120, 106, 2, 607, 61, 420, 811, 29, 125, 14, 20, 37, 105, 28, 248, 16, 159, 7, 35, 19, 301, 125, 110, 486, 287, 98, 117, 511, 62, 51, 220, 37, 113, 140, 807, 138, 540, 8, 44, 287, 388, 117, 18, 79, 344, 34, 20, 59, 511, 548, 107, 603, 220, 7, 66, 154, 41, 20, 50, 6, 575, 122, 154, 248, 110, 61, 52, 33, 30, 5, 38, 8, 14, 84, 57, 540, 217, 115, 71, 29, 84, 63, 43, 131, 29, 138, 47, 73, 239, 540, 52, 53, 79, 118, 51, 44, 63, 196, 12, 239, 112, 3, 49, 79, 353, 105, 56, 371, 557, 211, 505, 125, 360, 133, 143, 101, 15, 284, 540, 252, 14, 205, 140, 344, 26, 811, 138, 115, 48, 73, 34, 205, 316, 607, 63, 220, 7, 52, 150, 44, 52, 16, 40, 37, 158, 807, 37, 121, 12, 95, 10, 15, 35, 12, 131, 62, 115, 102, 807, 49, 53, 135, 138, 30, 31, 62, 67, 41, 85, 63, 10, 106, 807, 138, 8, 113, 20, 32, 33, 37, 353, 287, 140, 47, 85, 50, 37, 49, 47, 64, 6, 7, 71, 33, 4, 43, 47, 63, 1, 27, 600, 208, 230, 15, 191, 246, 85, 94, 511, 2, 270, 20, 39, 7, 33, 44, 22, 40, 7, 10, 3, 811, 106, 44, 486, 230, 353, 211, 200, 31, 10, 38, 140, 297, 61, 603, 320, 302, 666, 287, 2, 44, 33, 32, 511, 548, 10, 6, 250, 557, 246, 53, 37, 52, 83, 47, 320, 38, 33, 807, 7, 44, 30, 31, 250, 10, 15, 35, 106, 160, 113, 31, 102, 406, 230, 540, 320, 29, 66, 33, 101, 807, 138, 301, 316, 353, 320, 220, 37, 52, 28, 540, 320, 33, 8, 48, 107, 50, 811, 7, 2, 113, 73, 16, 125, 11, 110, 67, 102, 807, 33, 59, 81, 158, 38, 43, 581, 138, 19, 85, 400, 38, 43, 77, 14, 27, 8, 47, 138, 63, 140, 44, 35, 22, 177, 106, 250, 314, 217, 2, 10, 7, 1005, 4, 20, 25, 44, 48, 7, 26, 46, 110, 230, 807, 191, 34, 112, 147, 44, 110, 121, 125, 96, 41, 51, 50, 140, 56, 47, 152, 540, 63, 807, 28, 42, 250, 138, 582, 98, 643, 32, 107, 140, 112, 26, 85, 138, 540, 53, 20, 125, 371, 38, 36, 10, 52, 118, 136, 102, 420, 150, 112, 71, 14, 20, 7, 24, 18, 12, 807, 37, 67, 110, 62, 33, 21, 95, 220, 511, 102, 811, 30, 83, 84, 305, 620, 15, 2, 10, 8, 220, 106, 353, 105, 106, 60, 275, 72, 8, 50, 205, 185, 112, 125, 540, 65, 106, 807, 138, 96, 110, 16, 73, 33, 807, 150, 409, 400, 50, 154, 285, 96, 106, 316, 270, 205, 101, 811, 400, 8, 44, 37, 52, 40, 241, 34, 205, 38, 16, 46, 47, 85, 24, 44, 15, 64, 73, 138, 807, 85, 78, 110, 33, 420, 505, 53, 37, 38, 22, 31, 10, 110, 106, 101, 140, 15, 38, 3, 5, 44, 7, 98, 287, 135, 150, 96, 33, 84, 125, 807, 191, 96, 511, 118, 40, 370, 643, 466, 106, 41, 107, 603, 220, 275, 30, 150, 105, 49, 53, 287, 250, 208, 134, 7, 53, 12, 47, 85, 63, 138, 110, 21, 112, 140, 485, 486, 505, 14, 73, 84, 575, 1005, 150, 200, 16, 42, 5, 4, 25, 42, 8, 16, 811, 125, 160, 32, 205, 603, 807, 81, 96, 405, 41, 600, 136, 14, 20, 28, 26, 353, 302, 246, 8, 131, 160, 140, 84, 440, 42, 16, 811, 40, 67, 101, 102, 194, 138, 205, 51, 63, 241, 540, 122, 8, 10, 63, 140, 47, 48, 140, 288

The third paper is as follows:

317, 8, 92, 73, 112, 89, 67, 318, 28, 96,107, 41, 631, 78, 146, 397, 118, 98, 114, 246, 348, 116, 74, 88, 12, 65, 32, 14, 81, 19, 76, 121, 216, 85, 33, 66, 15, 108, 68, 77, 43, 24, 122, 96, 117, 36, 211, 301, 15, 44, 11, 46, 89, 18, 136, 68, 317, 28, 90, 82, 304, 71, 43, 221, 198, 176, 310, 319, 81, 99, 264, 380, 56, 37, 319, 2, 44, 53, 28, 44, 75, 98, 102, 37, 85, 107, 117, 64, 88, 136, 48, 151, 99, 175, 89, 315, 326, 78, 96, 214, 218, 311, 43, 89, 51, 90, 75, 128, 96, 33, 28, 103, 84, 65, 26, 41, 246, 84, 270, 98, 116, 32, 59, 74, 66, 69, 240, 15, 8, 121, 20, 77, 89, 31, 11, 106, 81, 191, 224, 328, 18, 75, 52, 82, 117, 201, 39, 23, 217, 27, 21, 84, 35, 54, 109, 128, 49, 77, 88, 1, 81, 217, 64, 55, 83, 116, 251, 269, 311, 96, 54, 32, 120, 18, 132, 102, 219, 211, 84, 150, 219, 275, 312, 64, 10, 106, 87, 75, 47, 21, 29, 37, 81, 44, 18, 126, 115, 132, 160, 181, 203, 76, 81, 299, 314, 337, 351, 96, 11, 28, 97, 318, 238, 106, 24, 93, 3, 19, 17, 26, 60, 73, 88, 14, 126, 138, 234, 286, 297, 321, 365, 264, 19, 22, 84, 56, 107, 98, 123, 111, 214, 136, 7, 33, 45, 40, 13, 28, 46, 42, 107, 196, 227, 344, 198, 203, 247, 116, 19, 8, 212, 230, 31, 6, 328, 65, 48, 52, 59, 41, 122, 33, 117, 11, 18, 25, 71, 36, 45, 83, 76, 89, 92, 31, 65, 70, 83, 96, 27, 33, 44, 50, 61, 24, 112, 136, 149, 176, 180, 194, 143, 171, 205, 296, 87, 12, 44, 51, 89, 98, 34, 41, 208, 173, 66, 9, 35, 16, 95, 8, 113, 175, 90, 56, 203, 19, 177, 183, 206, 157, 200, 218, 260, 291, 305, 618, 951, 320, 18, 124, 78, 65, 19, 32, 124, 48, 53, 57, 84, 96, 207, 244, 66, 82, 119, 71, 11, 86, 77, 213, 54, 82, 316, 245, 303, 86, 97, 106, 212, 18, 37, 15, 81, 89, 16, 7, 81, 39, 96, 14, 43, 216, 118, 29, 55, 109, 136, 172, 213, 64, 8, 227, 304, 611, 221, 364, 819, 375, 128, 296, 1, 18, 53, 76, 10, 15, 23, 19, 71, 84, 120, 134, 66, 73, 89, 96, 230, 48, 77, 26, 101, 127, 936, 218, 439, 178, 171, 61, 226, 313, 215, 102, 18, 167, 262, 114, 218, 66, 59, 48, 27, 19, 13, 82, 48, 162, 119, 34, 127, 139, 34, 128, 129, 74, 63, 120, 11, 54, 61, 73, 92, 180, 66, 75, 101, 124, 265, 89, 96, 126, 274, 896, 917, 434, 461, 235, 890, 312, 413, 328, 381, 96, 105, 217, 66, 118, 22, 77, 64, 42, 12, 7, 55, 24, 83, 67, 97, 109, 121, 135, 181, 203, 219, 228, 256, 21, 34, 77, 319, 374, 382, 675, 684, 717, 864, 203, 4, 18, 92, 16, 63, 82, 22, 46, 55, 69, 74, 112, 134, 186, 175, 119, 213, 416, 312, 343, 264, 119, 186, 218, 343, 417, 845, 951, 124, 209, 49, 617, 856, 924, 936, 72, 19, 28, 11, 35, 42, 40, 66, 85, 94, 112, 65, 82, 115, 119, 236, 244, 186, 172, 112, 85, 6, 56, 38, 44, 85, 72, 32, 47, 63, 96, 124, 217, 314, 319, 221, 644, 817, 821, 934, 922, 416, 975, 10, 22, 18, 46, 137, 181, 101, 39, 86, 103, 116, 138, 164, 212, 218, 296, 815, 380, 412, 460, 495, 675, 820, 952

The 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:

1. Assign to each word of the text a number in order from the first to the last word.
2. Extrapolate the first letter of each word using the numbers contained in the cipher.
3. 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(8) becomes(9) necessary(10) for(11) one(12) people(13) to(14) dissolve(15) the(16) political(17) bands(18) which(19) have(20) connected(21) them(22) with(23) another(24) and(25) to(26) assume(27) among(28) the(29) powers(30) of(31) the(32) earth(33) the(34) separate(35) and(36) equal(37) station(38) to(39) which(40) the(41) laws(42) of(43) nature(44) and(45) of(46) nature's(47) god(48) entitle(49) them(50) a(51) decent(52) respect(53) to(54) the(55) opinions(56) of(57) mankind(58) requires(59) that(60) they(61) should(62) declare(63) the(64) causes(65) which(66) impel(67) them(68) to(69) the(70) separation(71) we(72) hold(73) these(74) truths(75) to(76) be(77) self(78) evident(79) that(80) all(81) men(82) are(83) created(84) equal(85) that(86) they(87) are(88) endowed(89) by(90) their(91) creator(92) with(93) certain(94) unalienable(95) rights(96) that(97) among(98) these(99) are(100) life(101) liberty(102) and(103) the(104) pursuit(105) of(106) happiness(107) that(108) to(109) secure(110) these(111) rights(112) governments(113) are(114) instituted(115) ...

The 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, 140, 47, 29, 107, 79, 84, 56, 239, 10, 26, 811, 5, 196, 308, 85, 52, 160, 136, 59, 211, 36, 9, 46, 316, 554, 122, 106, 95, 53, 58, 2, 42, 7, 35...

The 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*). For example:

- *115 = instituted = I*
- *73 = hold = h*
- *24 = another = a*
- *807 (missing) = v*
- *37 = equal = e*
- *52 = decent = d*
- *49 = entitle = e*

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 goes from 1 to infinity, assuming that the attacker works with brute force and explores 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 ciphers are also interesting because this kind of algorithm could have new applications in modern cryptography 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 9*, *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 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. Another requirement is that the key has to be the same length as the message. These features make 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 key of the same length as the message avoids the problem of short messages, as we will see later. Finally, the key has to be completely random.

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 **Exclusive Or** (**XOR**), many times throughout this book, especially when we discuss symmetric encryption in *Chapter 2*, *Symmetric Encryption Algorithms*. 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)

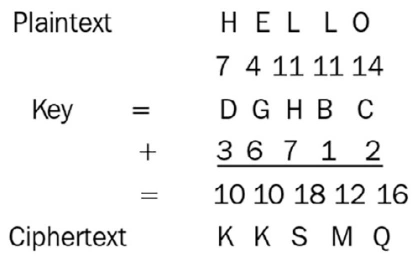
This Vernam cipher method follows four steps:

1. Transform the plaintext into a string of bits using ASCII code.
2. Generate a random key of the same length as the plaintext.
3. Encrypt the message by adding modulo 2 (**XOR**) of the plaintext bitwise to the key and obtain the ciphertext.
4. Decrypt the message by doing the inverse operation of adding the ciphertext to the key and obtain the plaintext again.

To use 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:



```
Plaintext         H  E  L  L  O
                  7  4  11 11 14
Key        =      D  G  H  B  C
           +      3  6  7  1  2
           =      10 10 18 12 16
Ciphertext        K  K  S  M  Q
```

Figure 1.10: Encryption scheme in the Vernam algorithm

So, after transposing the letters, the encryption of *[HELLO]* is *(KKSMQ)*.

You can do an exercise by yourself to decrypt the *(KKSMQ)* ciphertext with the Vernam cipher using the inverse process: applying *f[-K]* to the ciphertext, returning the *[HELLO]* plaintext.

One of the attacks that many algorithms suffer is 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. With data traffic and frequency analysis, I intend to study the use of the letters or groups of letters in a ciphertext. For example, the letter *e* is one of the most-used in the English language, so we can plan an attack based on the frequency of this letter. In other words, suppose that if we analyze the traffic of encrypted data and encounter a frequent recurrence of a character in a ciphertext, it could be referred to as the letter *e*.

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 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 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 Vernam cipher candidate at the **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 analyze.

To make a similitude, we can say that, in cryptography, "a weak link in 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 not be useful.

Let's go deeper to analyze some of the principles universally recognized in cryptography. The first statement is as follows: *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 for this 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 as follows: *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. Take the following as a simple example:

$$21 = 3 * 7$$

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 Algorithms*, when I will explain asymmetric encryption. But here, it is sufficient to explain why RSA is used to protect financial secrets, intelligence secrets, and other kinds of very sensitive secrets.

The reason for this 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 Inventions in Cryptography and Logical Attacks*, 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 by 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 or thousands of digits.

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 several machines and an optimization algorithm that elevated their computation.

The third statement is as follows: *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, for 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's 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 a cryptosystem having perfect secrecy and being 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 could 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-Hellman, and ElGamal) 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 9*, *Quantum Cryptography*. For now, I can say that classic 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 **AES** families, and also address some of the issues mentioned in this chapter.

Section 2

Classical Cryptography (Symmetric and Asymmetric Encryption)

This section will deeply analyze classical cryptography: symmetric and asymmetric encryption, hash functions, and digital signatures. It will guide you through the most famous algorithms used in cybersecurity and ICT.

This section comprises the following chapters:

- *Chapter 2*, *Symmetric Encryption Algorithms*
- *Chapter 3*, *Asymmetric Encryption Algorithms*
- *Chapter 4*, *Hash Functions and Digital Signatures*

2

Symmetric Encryption Algorithms

After covering 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 Dive into Cryptography*, we saw some symmetric cryptosystems such as **ROT13** and the **Vernam cipher**. Before going further into describing modern symmetric algorithms, we need to overview the construction of the classic 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:

- The basics of Boolean logic
- The basics of a simplified **Data Encryption Standard** (**DES**) where we start to familiarize ourselves with the techniques of S-box, substitution, and transposition of data

- Analyzing DES, Triple DES, and DESX by applying the previously mentioned techniques to these algorithms
- The **Advanced Encryption Standard** (**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 understand 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 Dive 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 **AND** (conjunction**), OR** (disjunction), and **XOR** (exclusive **OR**). 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 **TRUE** or **FALSE** condition. This problem is called the **Boolean satisfiability problem** (**SAT**, or **B-SAT**) and it is of particular importance in computer science. SAT was the first problem to be shown as **NP-complete**.

NP-complete refers to the classical NP problem in the **theory of complexity**. If a group of questions is answerable in a reasonable time, we say **P** for **polynomial time**. If the time of answering is **NP** (for **nondeterministic polynomial**), then we say that this group of questions is not tractable in a reasonable running machine time. These questions are therefore NP-complete. So, in general, this is a hard problem to solve.

Conversely, I say that this is a hard problem only for a classical computer. An example is the **RSA problem of factorization of a semiprime**, which can be characterized as an NP problem. We will see that RSA theoretically will not be an issue for a quantum computer applying an appropriate quantum algorithm with a proper number of qubits (*Chapter 9*, *Quantum Cryptography*).

The question now is as follows: given a certain function, does an assignment of the **TRUE** or **FALSE** values 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*, *Zero-Knowledge Protocols*, when we talk about **zero knowledge**, and *Chapter 8*, *Homomorphic Encryption and Crypto Search Engine*, where we talk about a search engine that works with encrypted data:

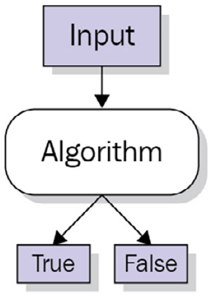


Figure 2.1: A Boolean circuit gives two opposite variables as output

The basic operations performed in Boolean circuits are as follows:

- **AND** (**conjunction**): Denoted with the symbol $x \wedge y$. 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 through 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:

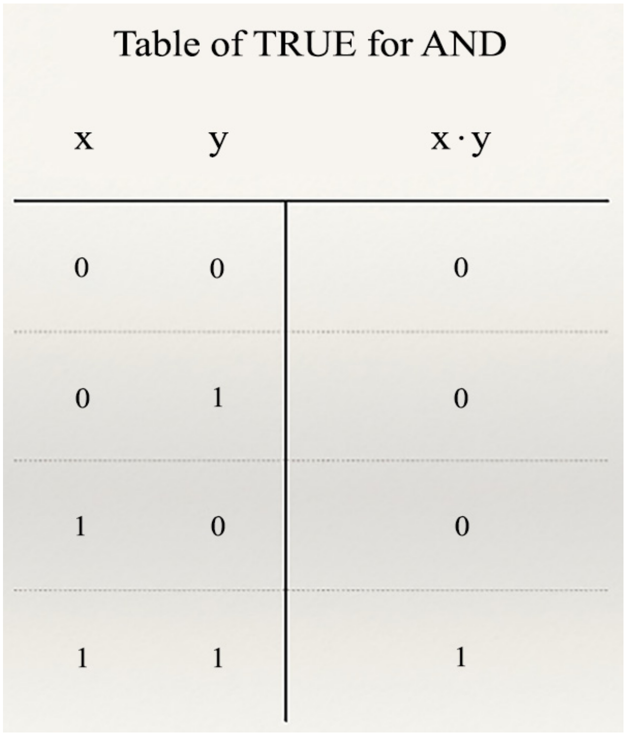


Table of TRUE for AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

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:

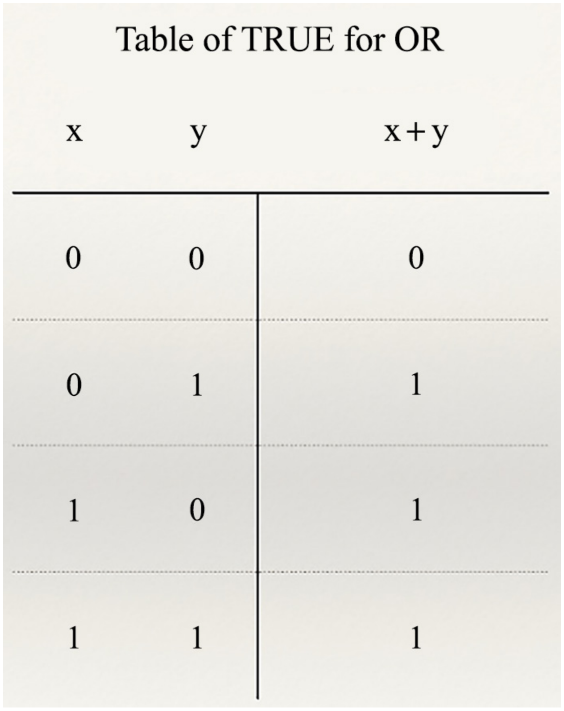


Table of TRUE for OR

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

Figure 2.3: Mathematical table for OR

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 $\neg x$, 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 the **NOT** operator denoted by the symbol of negation, $\neg x$. It is represented by a unitary operation that gets back the opposite value with respect to its input:

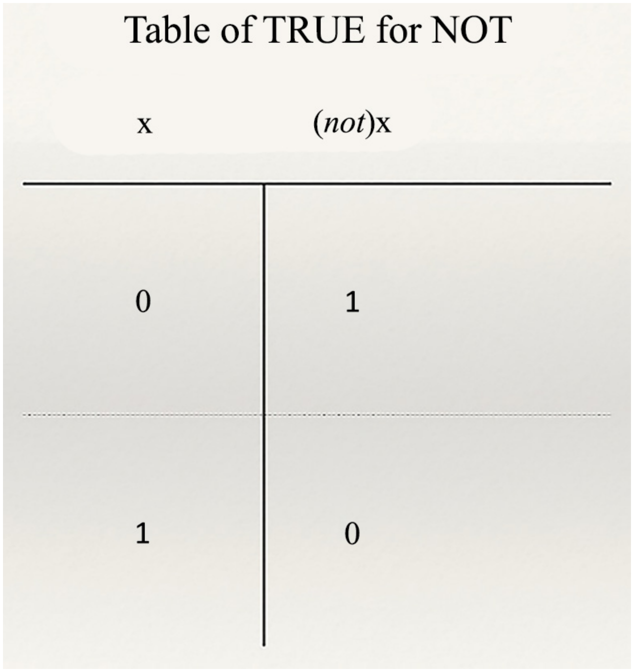


Table of TRUE for NOT

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:

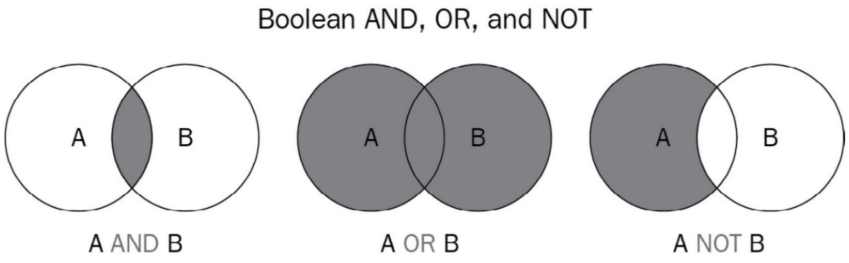


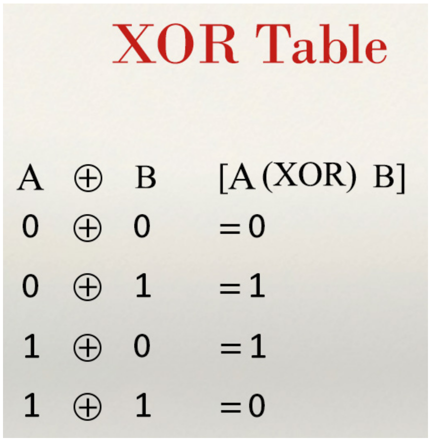
Figure 2.5: Boolean operators represented by a Venn diagram

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.

XOR is also denoted by the $\oplus$ symbol.

The operation of $A \oplus B$ gives back the logic value of **1** if the number of variables that assume a value of **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 1s 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 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 **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 cipher**, 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 at 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 4 x 16 matrix consisting of 6 bits as input and 4 bits as output, which introduces non-linear mapping between the input and output and causes confusion in the data.

We will find that the S-box is present in all modern symmetric encryption algorithms, such as DES, Triple DES, Blowfish (and the more modern Twofish), and AES.

The four rows are represented by progressive 2 bits, as follows:

00

01

10

11

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 an S-box is implemented and how it works, here is an example: **011011**. This string of bits has two outer bits, **0** and **1**, and four middle bits, **1101**.

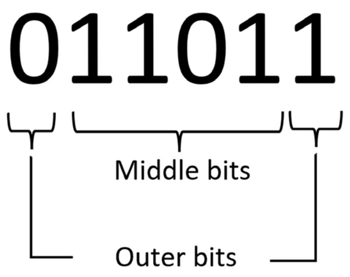


Figure 2.7: String bits when implementing an S-box

So, let's take the outer bits to form **01** and take the middle bits as **1101**. Working in the binary system, using *N2* notation, **(01)2** corresponds to the 2nd row in the matrix, and $\mathbf{(1101)_2}$ corresponds to the 13th column. You can see the representation in binary numbers of the S-box matrix described here:

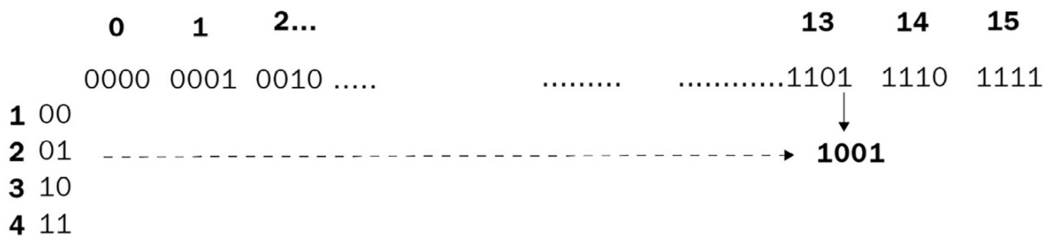
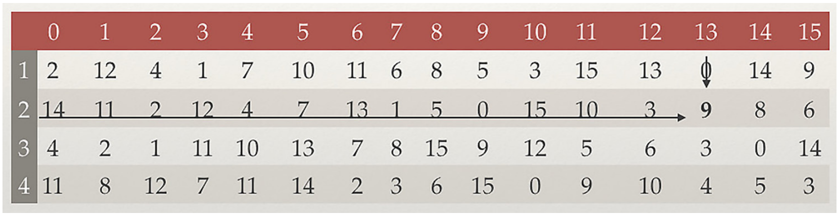


Figure 2.8: An S-box matrix (intersection) of 4 x 16 represented in binary numbers

As you can see, by finding the intersection of the column and the row, we obtain **$(1001)_2$**.

The same matrix can be represented in decimal numbers:



	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 2.9: An S-box matrix of 4 x 16 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 $\mathbf{R_0}$, the right half, consists of the last 6 bits:

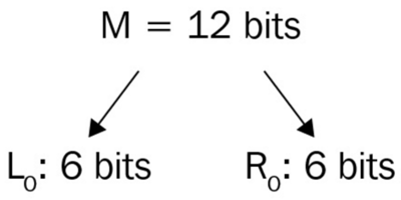


Figure 2.10: Message (M) is split into 6 bits to the left and 6 bits to the right

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. As we will see, L_0 and R_0 have the same bit expansion pattern: they are simply a division of M.

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 Code Book** (**ECB**) to divide the 64 bits of plaintext into 8 × 8 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:

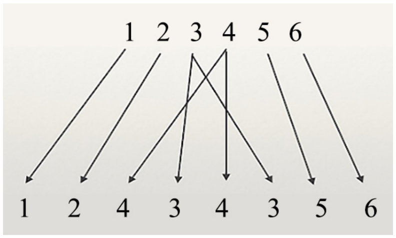


Figure 2.11: Bit expansion function

As you can see in the preceding figure, **L_0 = $(011001)_2$** has been expanded with **f [12434356]**.

Then, **L_0: 011001** will be expanded into **$(01010101)_2$**, as shown in the following figure:

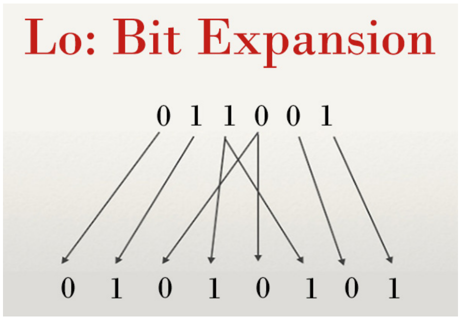


Figure 2.12: L_0 $(011001)_2$ bit expansion 8 bits

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

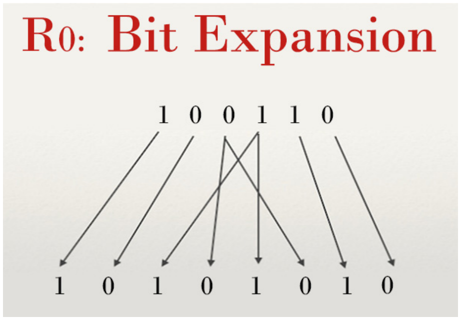


Figure 2.13: R_0 $(100110)_2$ bit expansion 8 bits

So, the expansion of **R_{i-1}** will be **$(10101010)_2$**.

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 **K_4** (related to the fourth round):

- **K = 010011001** (9-bit key, the master key)
- **K_4 = 01100101** (8 bits taken by **K**)

The following figure will help you better understand the process:

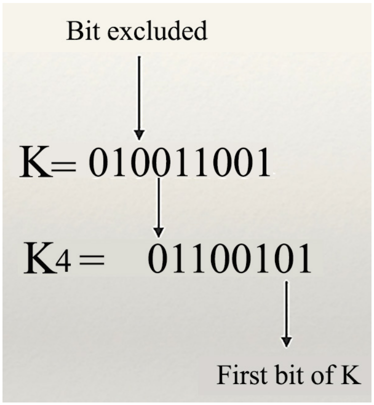


Figure 2.14: Example of key generation

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 the master key **[K]** to generate **[K4]**.

Bit encryption

To perform the bit encryption, **(E)**, we use the **XOR** function between **R_{i-1}= (100110)2** expanded and **K_i = $(01100101)_2$**. See *Figure 2.6* for how to use XOR.

I call this output **$E(K_i)$**:

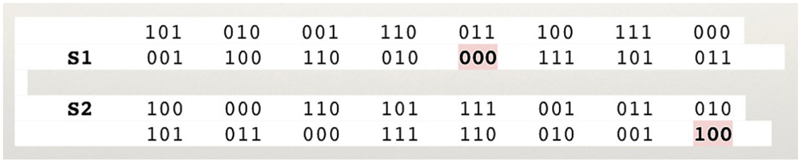
$$Exp(R_{i-1}) \oplus E(K_i)(11001111)$$

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:

$$L(EK_i) = (1100)_2 \qquad R(EK_i) = (1111)_2$$

Now, we process the 4 bits to the left and the 4 bits to the right with two S-box 2 x 8 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 2.15: Examples 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:

$$S1\left(L\left(E(K_i)\right)\right) = (000)_2 \parallel S2\left(R\left(E(K_i)\right)\right) = (100)_2$$

$$000 \parallel 100 = (000100)_2 f(R_{i-1}, K_i) = (000100)_2$$

The following figure shows how the encryption of the first round (the right side) of the *f* function mathematically works:

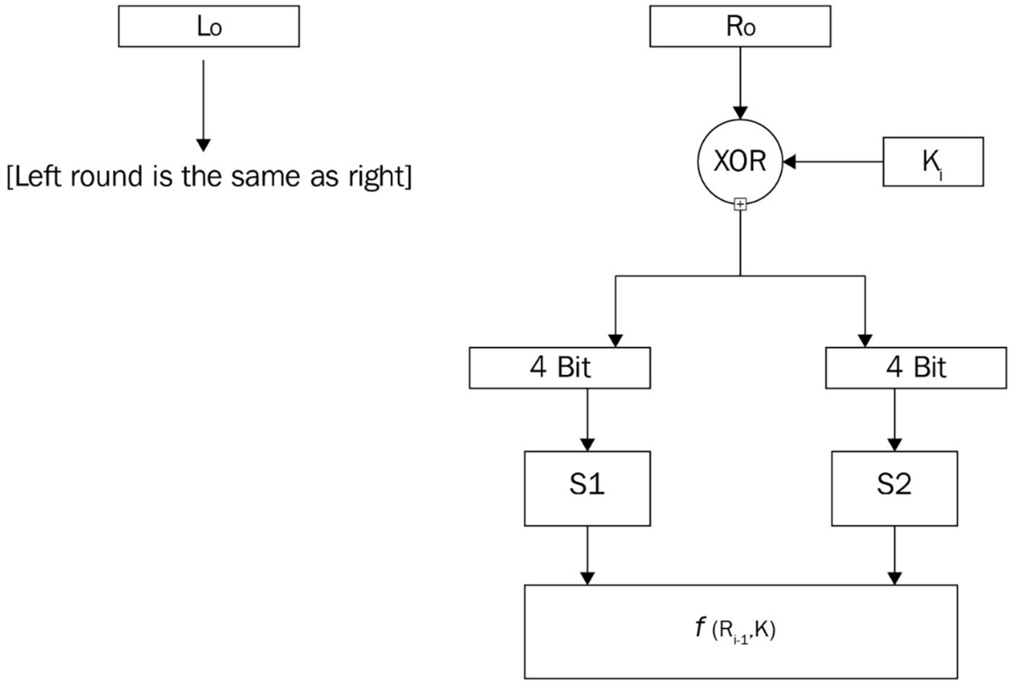


Figure 2.16: Mathematical scheme of (simple) DES encryption at the first round (right side)

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 works.

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.

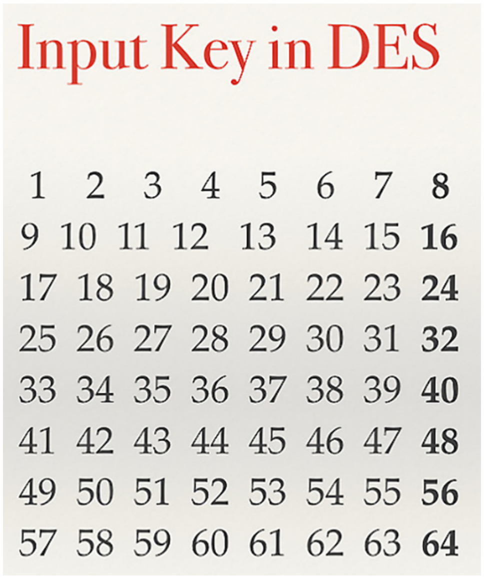
I would like you to focus on the DES encryption scheme of *Figure 2.15* to fully understand DES encryption.

Key generation in DES

In 1945, Claude Shannon introduced the principles of *confusion* and *diffusion* in his paper, *A Mathematical Theory of Cryptography* (`https://www.iacr.org/museum/shannon45.html`). DES, just like most symmetric algorithms, uses bit scrambling to obtain these two effects.

In Shannon's original definitions, **confusion** refers to *making the relationship between the ciphertext and the symmetric key as complex and involved as possible*, whereas **diffusion** refers to *dissipating the statistical structure of plaintext over the bulk of ciphertext*. This complexity is generally implemented through a well-defined and repeatable series of substitutions and permutations.

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

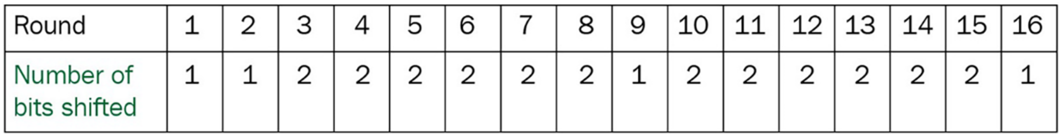
1	2	3	4	5	6	7	**8**
9	10	11	12	13	14	15	**16**
17	18	19	20	21	22	23	**24**
25	26	27	28	29	30	31	**32**
33	34	35	36	37	38	39	**40**
41	42	43	44	45	46	47	**48**
49	50	51	52	53	54	55	**56**
57	58	59	60	61	62	63	**64**

Figure 2.17: 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**. The first permutation is also known as the **initial permutation** and is a crucial step in DES.

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 2.18: Showing the number of key bits shifted in each round in DES

If you look at the preceding table, you can see that 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 **C0**, **D0** (the original division of the key in 28-bit left and 28-bit right), expressed in binary notation as follows:

$$C0 = 1111000011001100101010101001$$

$$D0 = 0101010101100110011110001111$$

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

$$C1 = 1110000110011001010101010011$$

$$D1 = 1010101011001100111100011110$$

If 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**:

$$C0 = 1111000011001100101010101001$$

$$C1 = 1110000110011001010101010011$$

After the circular shift, 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 2.19: 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**:

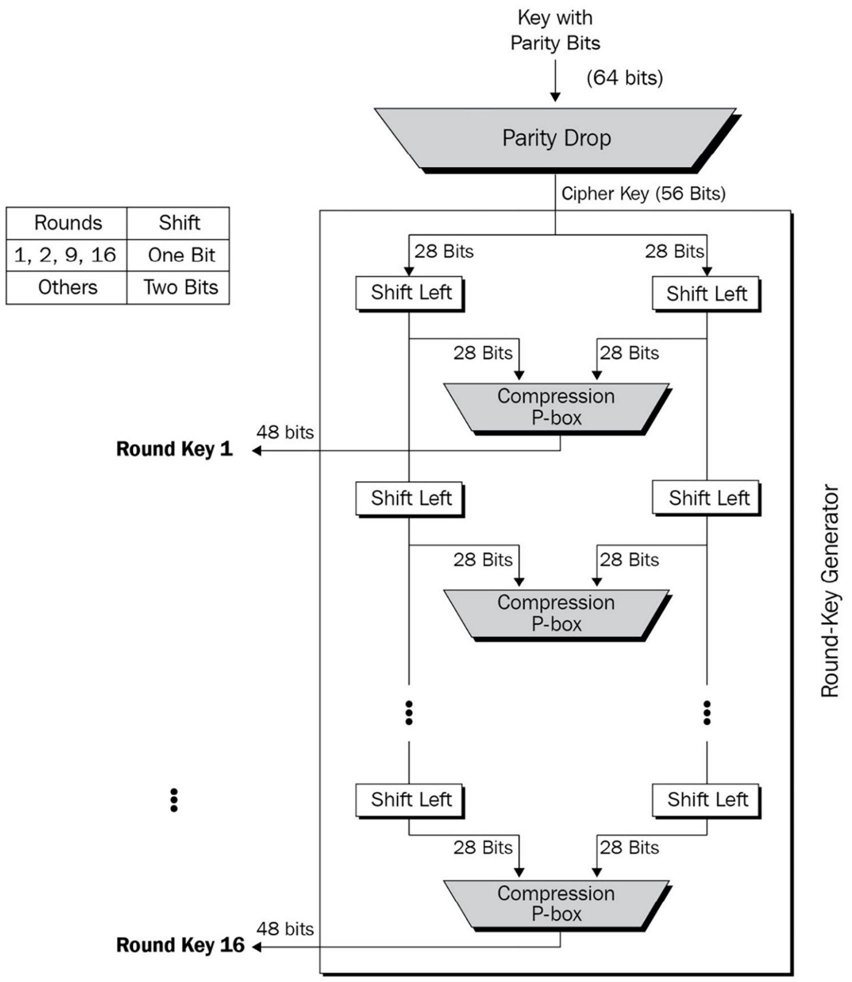


Figure 2.20: The key generation scheme

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:

1. **Initial permutation (IP)**: First, the bits of the message, **[M]**, are permutated through a function that we call **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.
2. **Rounds of encryption**: For $0 \leq i \leq 16$, the following operations are executed:

    a. Li = Ri-1

    b. Compute $R_i = L_i - 1 \oplus f(R_i - 1, K_i)$.

    Here, K_i 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.

    c. Basically, for **i = 1** (the first round), we have the following:

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f(R_0, K_1)$$

- **Final permutation**: The last part of the algorithm at the 16th round (the last one) consists of the following:

    a. Exchanging the left part, L_{16}, with the right part, R_{16}, in order to obtain (R_{16}, L_{16})

    a. Applying the inverse, *INV*, of the IP to obtain the ciphertext, *c*, where $c = INV(IP(R_{16}, L_{16}))$

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

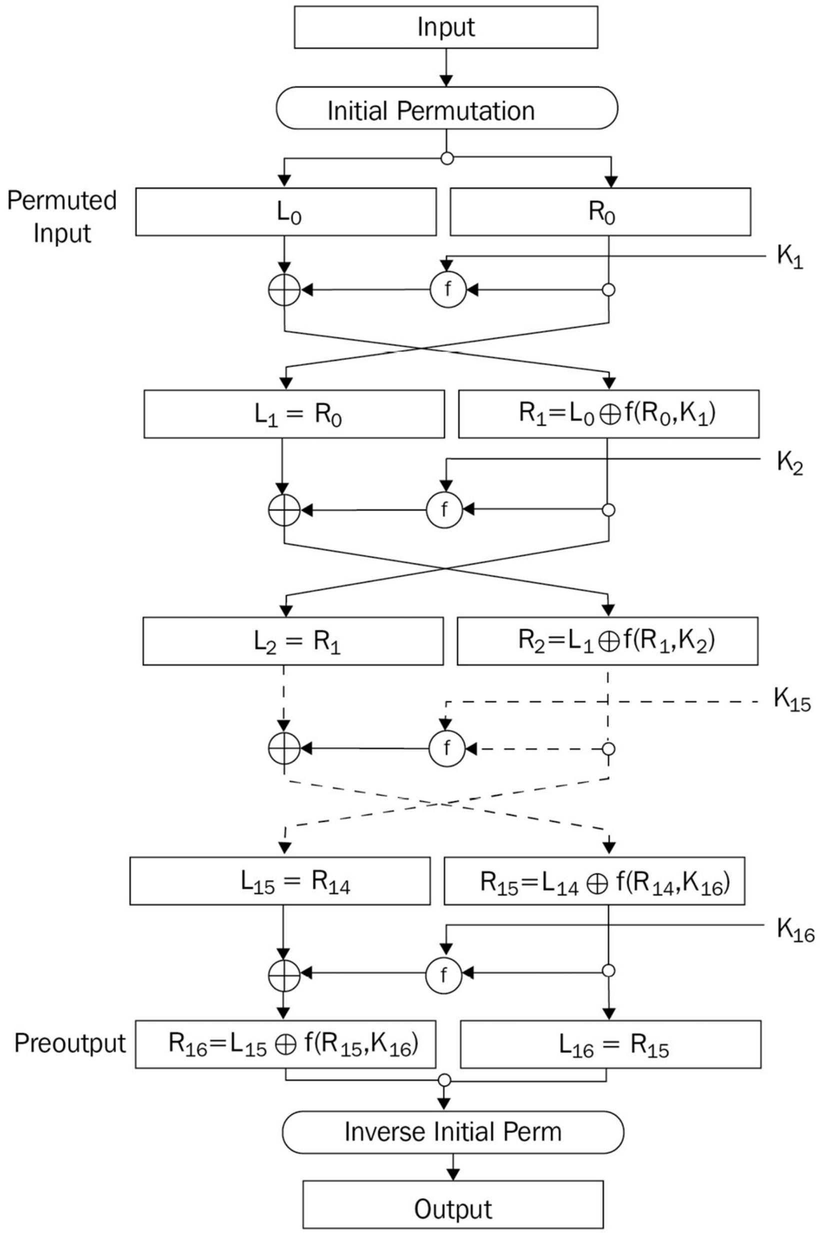


Figure 2.21: DES encryption

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 ($K_1 ... K_{16}$) so that it becomes ($K_{16} ... K_1$).

In the following figure, you see the decryption process in a flow chart scheme:

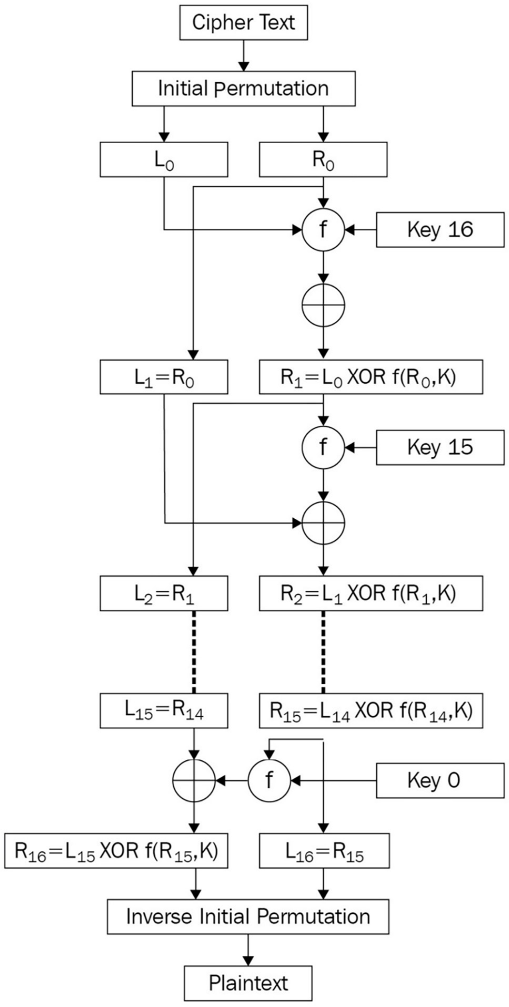


Figure 2.22: DES decryption process

So, to describe the decryption process: you take the ciphertext and operate the first IP on it, then *XOR* L_0 (left part) with $R_0 = f(R_0, K_{16})$.

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.

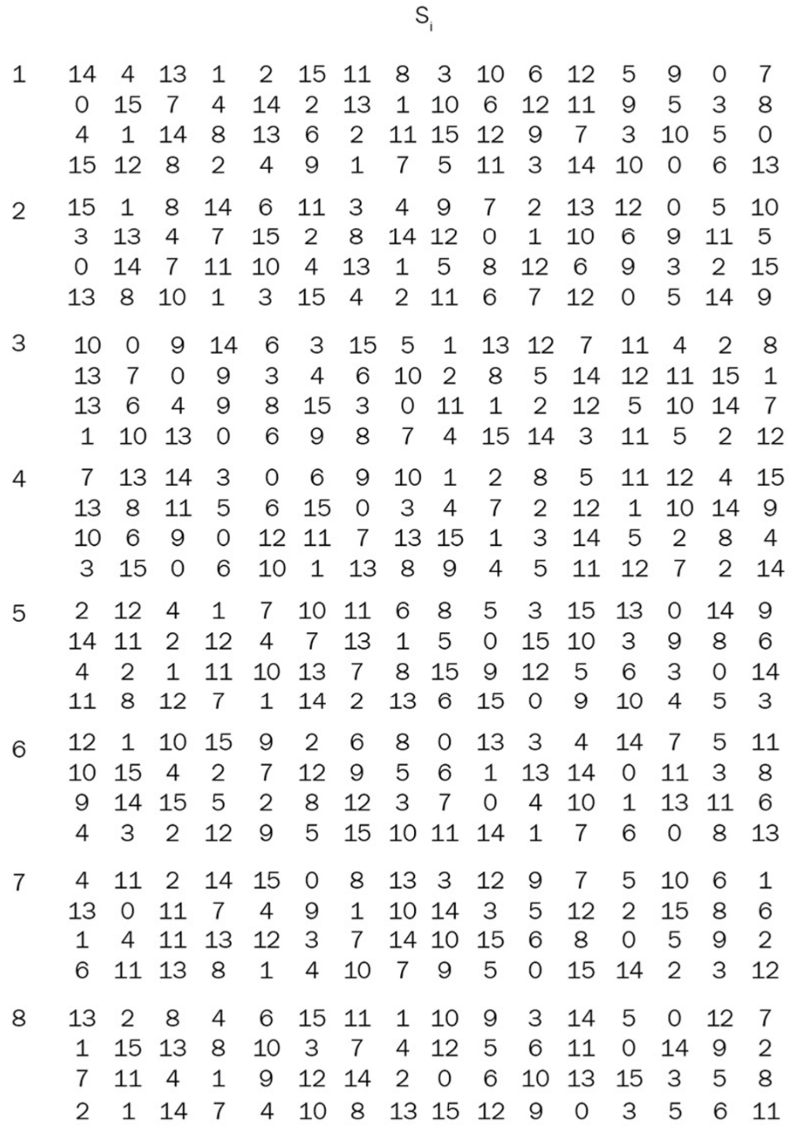
Analysis of the DES algorithm

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 ($K_1, K_2 \ldots K_{16}$) and the half part of the message, **[M]**, at each round.

In this step, we find the S-box: the *f* function 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 4×16) fixed by the NSA. In the following image, we can see eight rounds of the S-box:



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 2.23: The S-box matrix in DES

As you might notice, the numbers included are between **0** and **(R_{i-1}) 16-1 = 15**.

Take a look at the specifics of the S-box in the 5th round of DES:

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	11	14	2	13	6	15	0	9	10	4	5	3

Figure 2.24: S-box 5th round

If you observe carefully, in the 14th column, all the numbers are very low: **0**, **9**, **3**, and **4**. This combination could 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. Let's look into this.

A question that may come to you spontaneously might be: *Why is the key only 56 bits and not 64 bits?* This is because the other 8 bits are used for pairing.

Actually, the initial master key is 64 bits in length, so every 8th 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 2.25: Bit discarded in the DES key generation algorithm

Remember the step where we process a selection of 48 bits over the subset key of 56 bits. So, 48 bits of input will give exactly 48 bits of output after the **XOR** operation is performed with **[K_i]**.

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, exponential 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 Algorithms*) 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 the **DES cracker** to break DES for DES Challenge 2, aiming for the $10,000 reward for the decryption of the ciphertext. 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. DES Challenge 3 only took 22 hours and 15 minutes. The method adopted was a simple *brute-force* method to analyze all the possible combinations of bits given by the 256 (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: it would take 1 microchip 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 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. But we need to try less than all the sets of keys because, statistically, as proved by Mitsuru Matsui, with (247) 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 you are interested, you can find 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 the 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 (for clarity's sake, I should state that we are using binary, with no parity bits):

- 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:

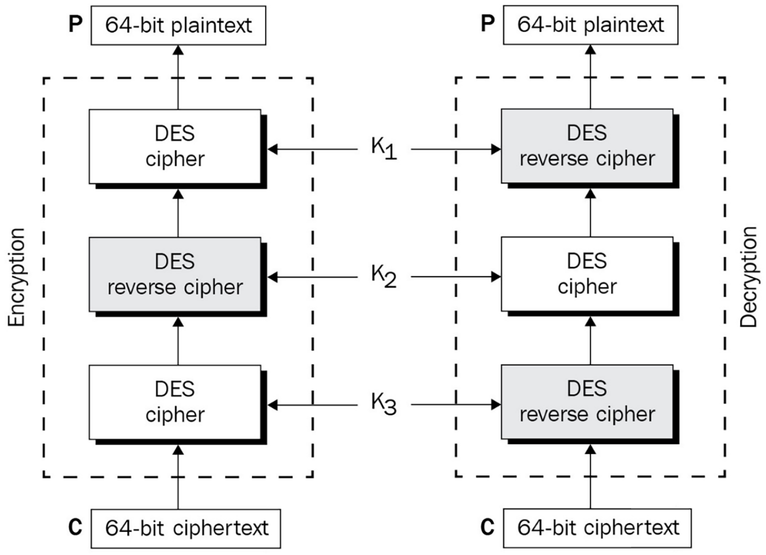


Figure 2.26: Triple DES encryption/decryption scheme

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:

1. Encrypt the plaintext blocks using single DES with the *[K_1]* key.
2. Now, decrypt the output of *Step 1* using single DES with the *[K_2]* key.
3. 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 co-author 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:

$$C = [K_3] \oplus EK_1([K_2] \oplus [M])$$

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 E_{K1}, *XOR*, and *[K$_3$]* outputs. This method allows us to increase the virtual key to *64 + 56 + 64 = 184* bits, instead of the normal 56 bits:

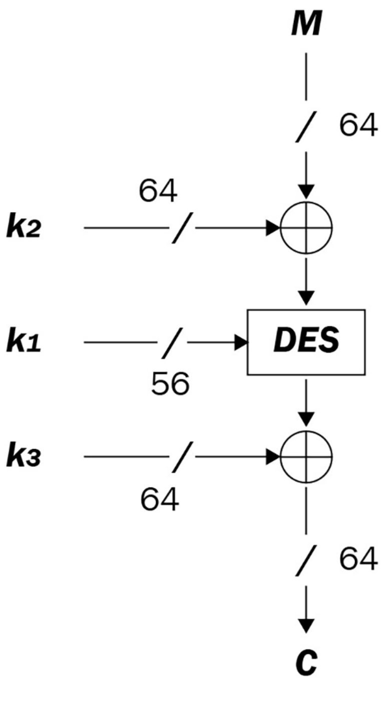


Figure 2.27: DESX encryption scheme

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

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**), **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 operate in blocks of 128 bits of input or it should be able to be used on different kinds of hardware, from 8-bit processors (also used in a smart card) to 32-bit architectures, commonly adopted in personal computers. Finally, it should be fast and very robust.

Under certain conditions (which 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 8*, *Homomorphic Encryption and 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** (**KE**) and **first add round key** (**F-ARK**); then, as you will see later on, each step is divided into four sub-steps, called **SubBytes** (**SB**) **transformation**, **ShiftRows** (**SR**) **transformation**, **MixColumns** (**MC**), and **AddRoundKey** (**ARK**). 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:

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

F-ARK works as follows.

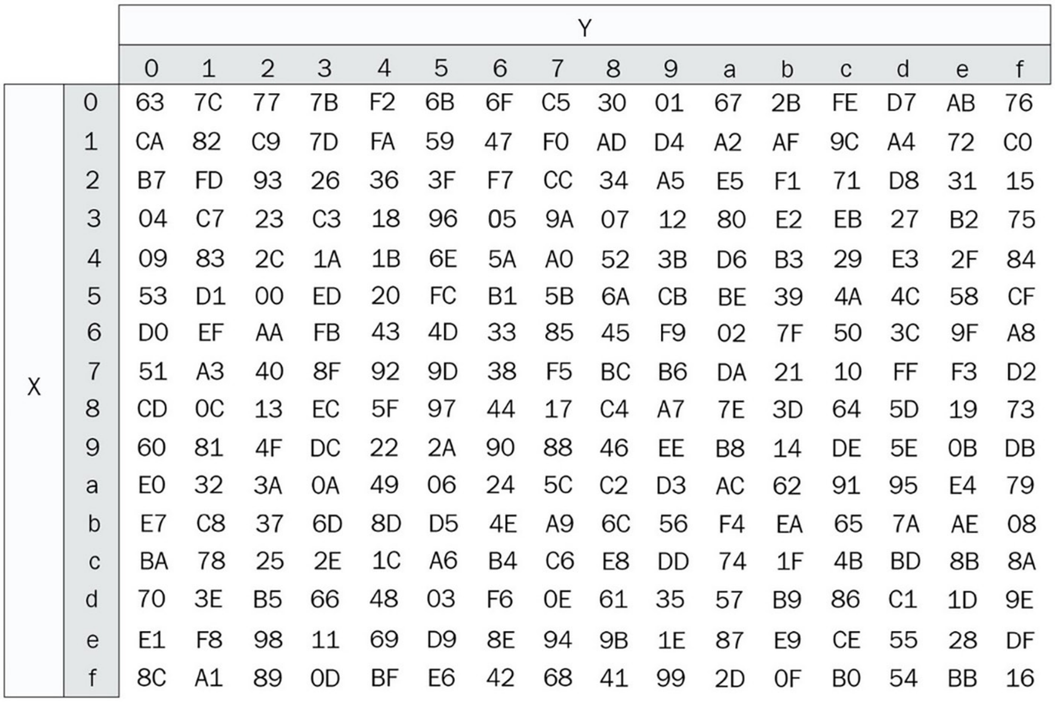
It is the first operation. The algorithm takes the first key, $[K_1]$, and adds it to **AddRoundKey:** using a bitwise **XOR** of the current block with a portion of the expanded key.

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:

1. **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.
2. **SR transformation**: This is a scrambling of a bit that causes diffusion on multiple rounds.
3. **MC transformation**: This step has a similar scope to SR but is applied to the columns.
4. **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
	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	05	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
X	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 2.28: S-box Rijndael

The last round runs all the operations of the previous rounds except for layer 3: **MC**. After all the previously 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:

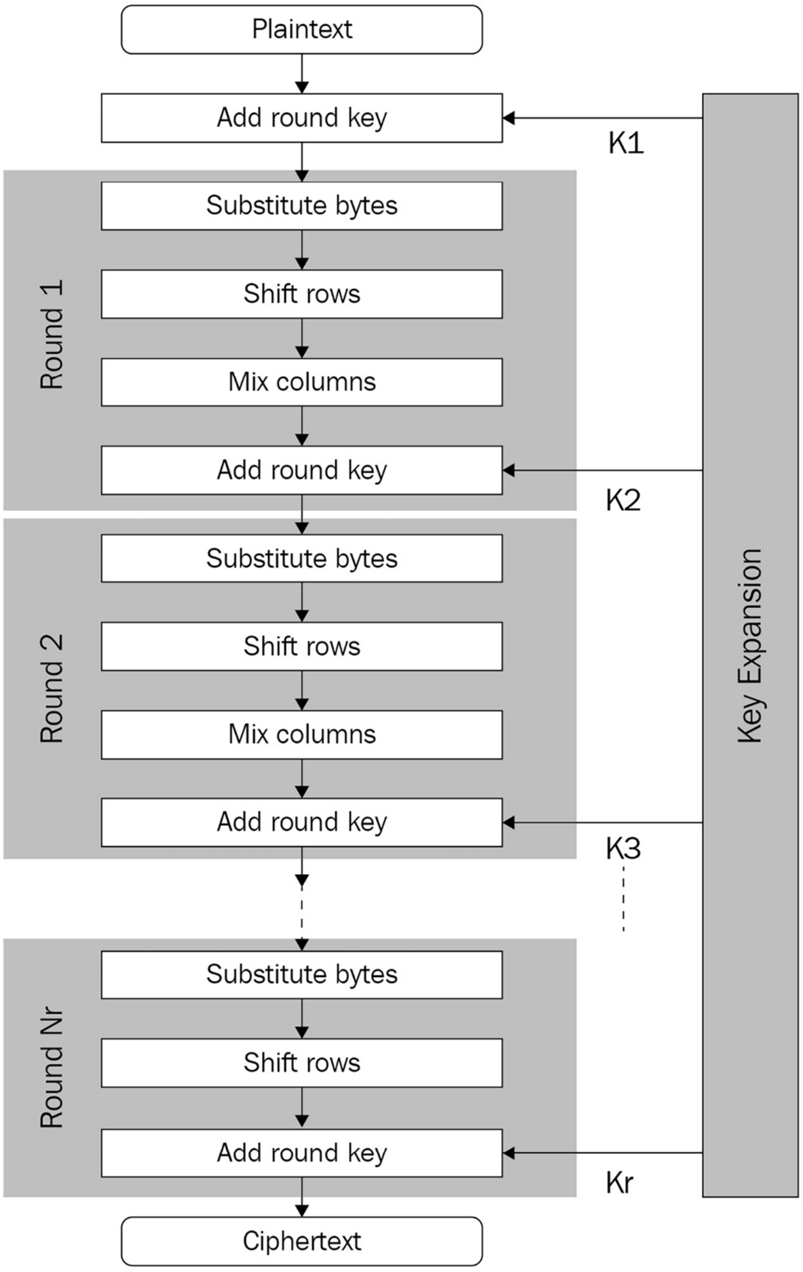


Figure 2.29: Encryption scheme in AES

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

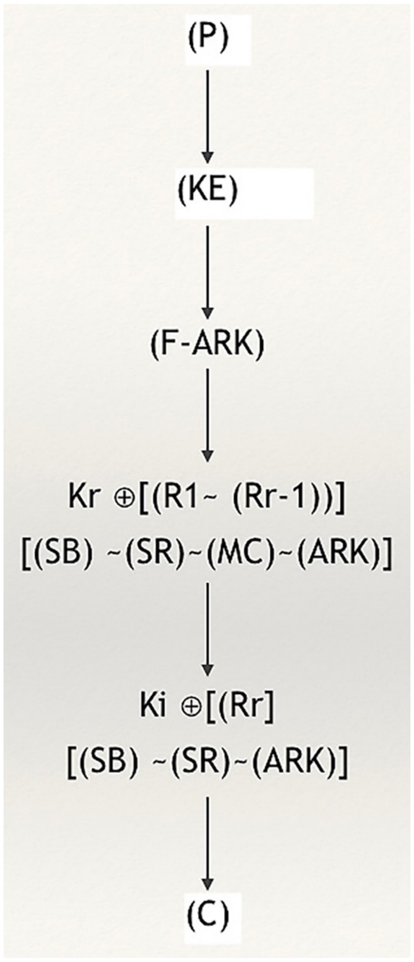


Figure 2.30: Re-schematizing an AES flow chart with mathematical functions

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

- **$K_r \oplus$ [(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 round, 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 see, *MC* is not present.

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

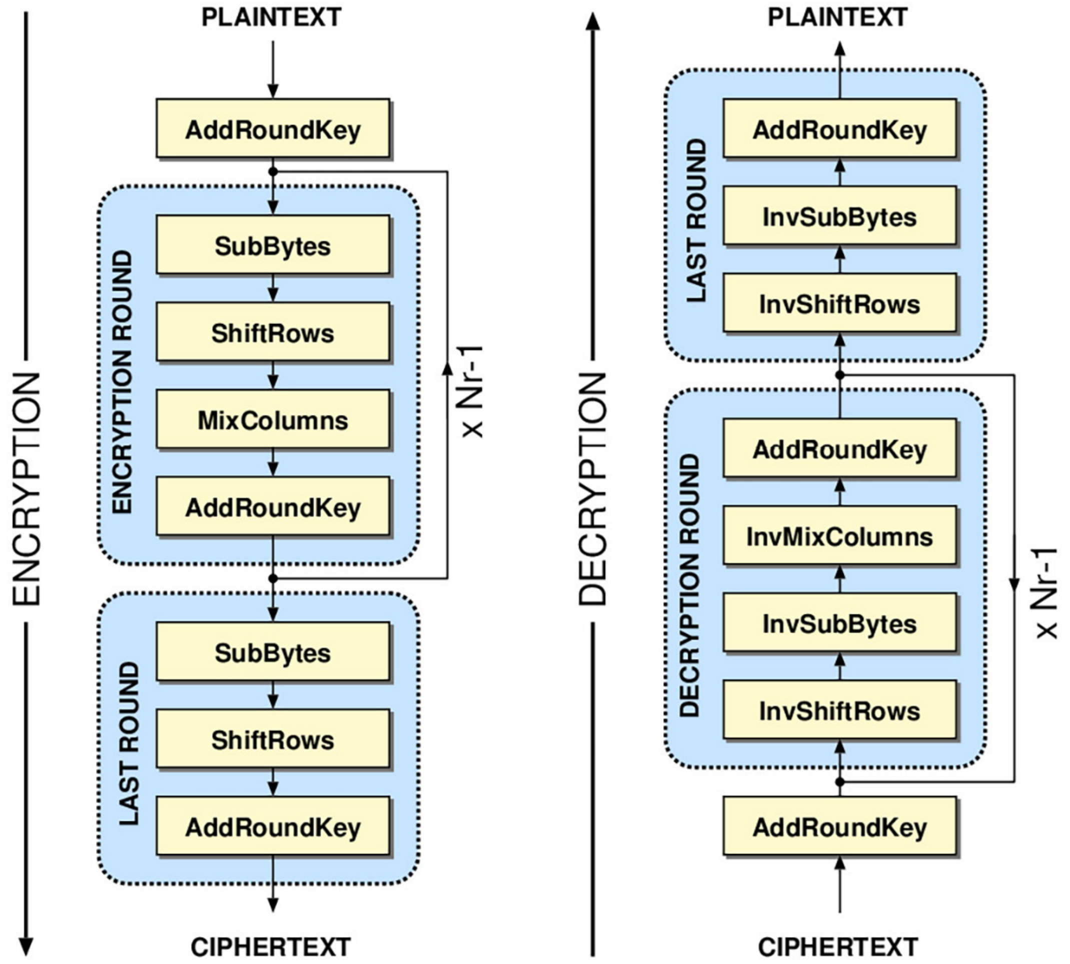


Figure 2.31: Encryption and decryption scheme in AES

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

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 8*, *Homomorphic Encryption and 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:

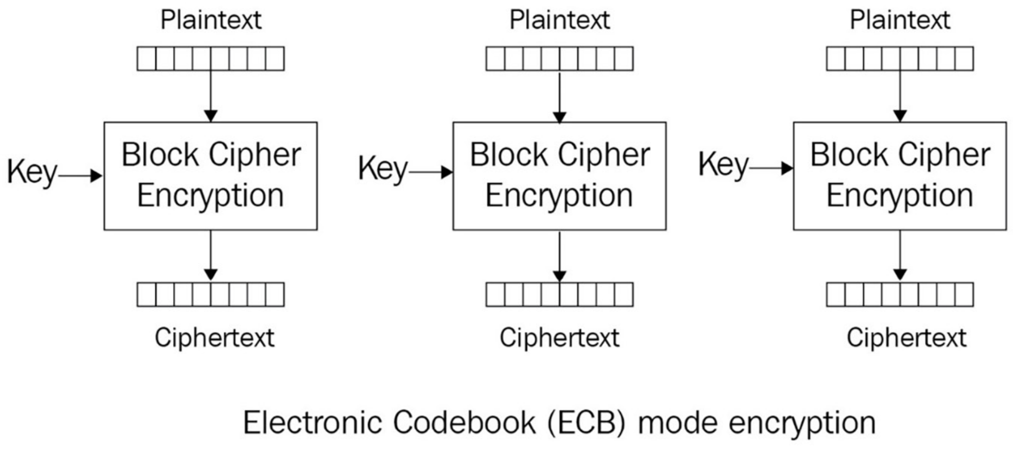


Figure 2.32: ECB mode encryption

If AES were implemented in ECB 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 (or in English, Matterhorn) mountain, it's possible to recognize the content even though it's encrypted:

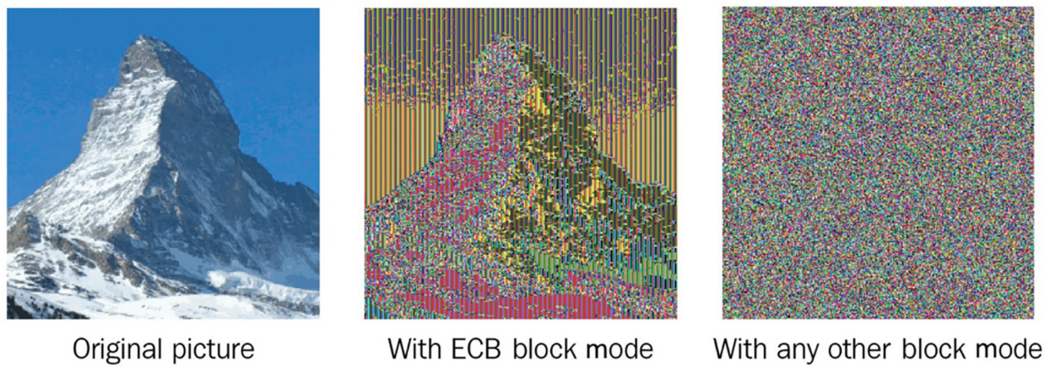


Figure 2.33: The original picture of Cervino mountain, encrypted with ECB (failure) mode, and encrypted with another block mode

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 resend 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:

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

$$CF\ \ A3\ \ 1C\ \ F4\ \ 67\ \ T3\ \ 2D\ \ M9 \ldots\ldots$$

3. 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.

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 8*, *Homomorphic Encryption and 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

The encryption works as follows:

1. Calculate the initial block, C_0, taking the first block of the plaintext, P_0, **XOR**-ing with **IV**:

$$C_0 = E(P_0 \oplus IV)$$

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

$$C_i = E(P_i \oplus C_{i-1})$$

Subsequently, the decryption works as follows:

1. To obtain the plaintext, P_0, combine **XOR** between the decryption of the first block of ciphertext received C_0 and **IV**:

$$P_0 = D(C_0) \oplus IV$$

2. 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:

$$P_i = D(C_i) \oplus C_{i-1}$$

In the following figure, the scheme of CBC mode encryption is represented:

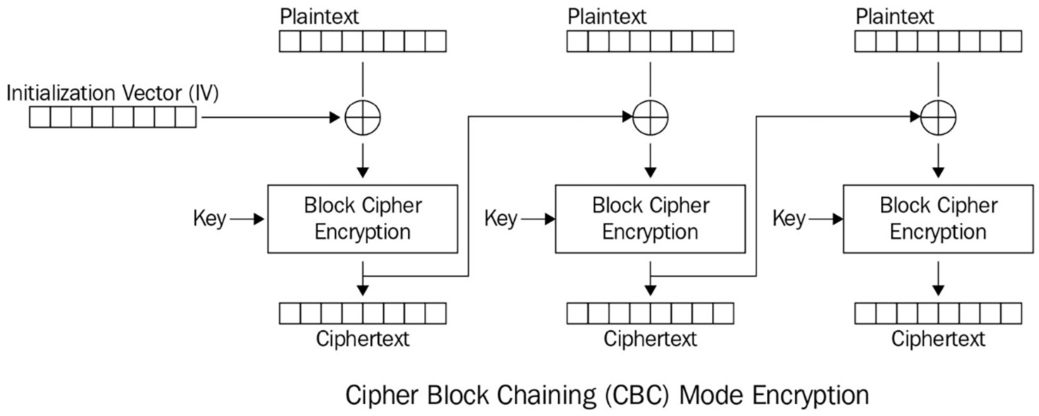


Figure 2.34: Scheme of CBC mode encryption

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, 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 and 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 a **field-programmable gate array** (**FPGA**), measuring the radiation that emanated from an antenna through an oscilloscope.

Finally, the real problem of AES is the exchanging of the key. With 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 8*, *Homomorphic Encryption and Crypto Search Engine*, when analyzing the CSE.

Summary

Now that we have explored some of the best symmetric algorithms and understood their peculiarities, a question originally posed in the introduction of this chapter is still open: *"But how do we implement symmetric algorithms that are robust (in the sense of security) and easy to perform (computationally) at the same time?"* One possible answer is the **AES algorithm**; it is robust and computationally easy at the same time. By learning about symmetric encryption, we have taken our first steps toward understanding this algorithm.

To start with, you learned about the basics of symmetric encryption. We have explored the Boolean operations necessary for understanding symmetric encryption, KE, and S-box functionality. Then, we took a deep dive 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 8*, *Homomorphic Encryption and 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 Algorithms

Asymmetric encryption involves 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 card numbers and online banking information; 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 began 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 the Cold War between the USA and the USSR was still ongoing. 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 block 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 or 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 a strange combination of introverts and 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 Hellman* 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, like 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 a *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 the **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 so 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 use 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 J. Watson* Research Center, 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 sceptical 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 the US 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 us 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 struggle 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 Hellman's support, found the solution, about which Hellman later said, *"God rewards fools!"*

Let's explore a mental representation of what Alice and Bob should do to exchange the key:

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.
2. Bob applies one more lock to the cage using his private key and sends the box back 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.
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, and 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.
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:

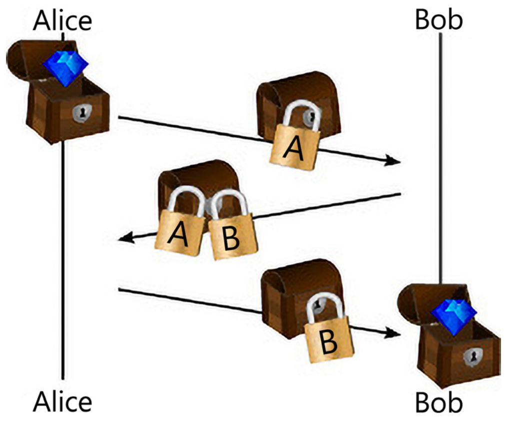


Figure 3.1: The D-H algorithm using the Alice and Bob example

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)*, lying 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*:

$$1 * 1 \equiv 1 \ (mod\ 7)$$

$$2 * 4 \equiv 1 \ (mod\ 7)$$

$$3 + 5 \equiv 1 \ (mod\ 7)$$

$$3 * 5 \equiv 1 \ (mod\ 7)$$

Therefore, 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*:

$$2 + 2 \equiv 1 \ (mod\ 3)$$

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 the *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 start working with a discrete logarithm.

So, let's say we have to solve an equation like this:

$$a^{[x]} \equiv b \ (mod\ p)$$

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 logarithms 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:

$$2^4 \equiv (x)\ (mod\ 13)$$

First, we calculate $2^4 = 16$, then divide *16 : 13 = 1* by the remainder of *3*, so *x = 3*.

The discrete logarithm is just the inverse operation:

$$2^{[y]} \equiv (x)\ (mod\ 13)$$

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:

$$2^{[y]} \equiv 3\ (mod\ 13)\ for\ [y]$$

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*:

$$2^{[4+(13-1)*1]} \equiv 2^{16} \ (mod\ 13)$$

$$2^{16} \equiv 3 \ (mod\ 13)$$

But it is also valid for *n = 2*:

$$2^{[4+(13-1)*2]} \equiv 2^{28} \ (mod\ 13)$$

$$2^{28} \equiv 3 \ (mod\ 13)$$

And so on...

Hence, the equation has infinite solutions for all the integers: that is, *(n ≥ 0)*:

$$[y] \equiv 2^{[4+12n]} \ (mod\ 13)$$

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 must 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, or key agreement 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*:

$$Symmetric\ Algorithm\ E\ ([k], M) = C$$

In other words, we use the D-H algorithm to generate the shared secret key, *[k]*, and 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 paragraphs.

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:

1. Alice and Bob first agree on the parameters: *(g)* as a generator in the ring, *(Zp)*, and a prime number, *p (mod p)*.

   Alice chooses a secret number, *[a]*, and Bob chooses a secret number, *[b]*.

   Alice calculates $A \equiv g^{a}$ *(mod p)* and Bob calculates $B \equiv g^{b}$ *(mod p)*.
2. Alice sends *(A)* to Bob and Bob sends *(B)* to Alice.
3. Alice computes $ka \equiv B^{a}$ *(mod p)* and Bob computes $kb \equiv A^{b}$ *(mod p)*.

So, *[ka = kb]* will be the secret that's shared between Alice and Bob.

The following is a numerical example of this algorithm:

1. Alice and Bob agree on the parameters they will use: that is, *g = 7* and *(mod 11)*.

   Alice chooses a secret number, *(3)*, and Bob chooses a secret number, *(6)*.

   Alice calculates 7^{3} *(mod 11)* $\equiv 2$ and Bob calculates 7^{6} *(mod 11)* $\equiv 4$.
2. Alice sends *(2)* to Bob and Bob sends *(4)* to Alice.
3. Alice computes 4^{3} *(mod 11)* $\equiv 9$ and Bob computes 2^{6} *(mod 11)* $\equiv 9$.

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

Analyzing the algorithm

In *Step 2* of the algorithm, Alice calculates $A \equiv g^{a}$ *(mod p)* and Bob calculates $B \equiv g^{b}$ *(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 section titled *The discrete logarithm*).

Another property of the modular powers is that we can write B^{a} and A^{b} *(mod p)*, as follows:

$$B^a \equiv (g^b)^a \ (mod\ p)$$

$$A^b \equiv (g^a)^b \ (mod\ p)$$

So, for the property of modular exponentiations, we have the following:

$$g^{(b*a)} \equiv g^{(a*b)} \ (mod\ p)$$

For example, we have the following:

- Alice: $(7^6)^3 \equiv 7^{(6*3)} \equiv 9 \ (mod\ 11)$
- Bob: $(7^3)^6 \equiv 7^{(3*6)} \equiv 9 \ (mod\ 11)$

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 (MitM)** attack.

A MitM attack is when the attacker infiltrates a channel of communication and spies on, blocks, or alters the communication between the sender and the receiver. Usually, the attack is accomplished by the attacker pretending to be one of the two true actors in the conversation. In our example, Eve is pretending to be Alice:

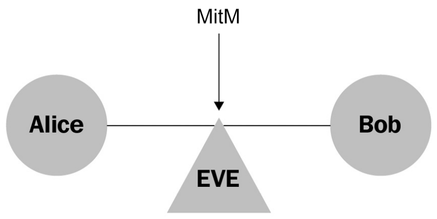


Figure 3.2: Eve is the "man in the middle"

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, and then Bob sends *(B)* to Alice. Now, Eve interferes with the communication by pretending to be Alice.

So, a MitM attack looks like this:

- **At** *Step 3*: Alice sends *(A)* to Bob and Bob sends *(B)* to Alice.
- **Here, we have the MitM attack**: Eve intercepts *(A)* and sends *(E)* to Bob. (Remember that *(E)* here represents Eve's public parameter, which has been generated by her private key, not encryption.) Then Bob sends *(B)* to Eve, assuming it is Alice.

Let's analyze Alice's function, $A \equiv g^a \ (mod\ p)$, and Bob's function, $B \equiv g^b \ (mod\ p)$.

This function would be crucial if it was done in normal arithmetic and not in finite fields. For example, 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 considered 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.

Proceeding with the final part of the algorithm, you can see the effects of the MitM trick.

Suppose that Alice and Bob are using D-H to generate a shared private key to encrypt the following 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 MitM 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.

This 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 MitM 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*, *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 MitM 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 MitM 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 a message inside a box, and lock it with the padlock.

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. We will revisit this idea in the next chapter.

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, Adi Shamir. 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 around midnight, Rivest went back home. He started to think over the problem that had been tormenting him for almost a year. Lying 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 Dive 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 computational 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:

$$[\boldsymbol{d}] * e \equiv 1 \ (mod \ [\boldsymbol{p-1}] * [\boldsymbol{q-1}])$$

Note

The elements in **bold** are protected and secret.

With this understanding, the key generation follows two steps:

Bob performs the encryption:

$$c \equiv \boldsymbol{M}^e \ (mod \ N)$$

Bob sends the ciphertext, *(c)*, to Alice. She can now decrypt *(c)* using her private key, *[d]*:

$$C^{\boldsymbol{d}} \equiv \boldsymbol{M}\ (mod\ N)$$

And that's it!

Numerical example

Let's use the following numbers:

- *M = 88*
- *e = 9007*
- *p = 101*
- *q = 67*
- *N = 6767*

We can now revisit the two steps from the key generation.

Bob's encryption is as follows:

$$88^{9007} \equiv 6621\ (mod\ 6767)$$

Alice receives a cryptogram, that is, *c = 6621*.

Alice's decryption is as follows:

$$9007 * d \equiv 1\ (mod\ (101 - 1) * (67 - 1)$$

$$d = 3943$$

$$6621^{3943} \equiv 88\ (mod\ 6767)$$

As 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:

$$\boldsymbol{M} \equiv c^{[d]}\ (mod\ N)$$

This is *Step 2* from the previous section: 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:

$$[d] * e \equiv 1\ (mod\ [p-1] * [q-1])$$

Here, *[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:

$$M^{(e*d)}\ (mod\ N)$$

$$de \equiv 1\ (mod\ (p-1) * (q-1))$$

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*.** We will see an algorithm in *Chapter 9*, *Quantum Cryptography*, that shows the way to break the RSA factorization problem. In the not-so-distant future, quantum computing will grow more powerful and certainly be the main adversary for classical cryptography.

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 9*, *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

Let's use the following numbers:

- *M = 2*
- *e = 3*
- *N = 77*
- $2^3 \equiv 8 \ (mod\ 77)$

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 Dive 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, which is then 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:

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

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

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

$$x^d \equiv y \ (mod\ N)$$

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:

$$y^e \equiv x \ (mod\ N)$$

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

$$x \equiv y^e \ (mod\ N)$$

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 $y^e \equiv x\ (mod\ N)$ equation is verified, Alpha can be confident that the data that's been sent from Beta is correct and that 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 can visualize the process by referring to the following diagram:

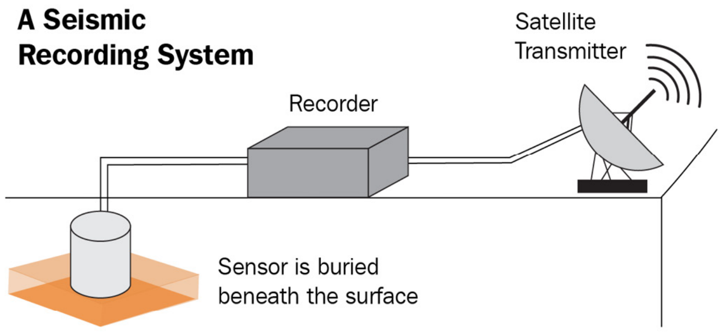

A Seismic Recording System
Satellite Transmitter
Recorder
Sensor is buried beneath the surface

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 Inventions in Cryptography and Logical Attacks*, 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, that 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 Inventions in Cryptography and Logical Attacks*.

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 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 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 semi-primes (*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 0, 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 101,000 digits.

Just to clarify how big such numbers are, you can consider that 1,080 represents the number of particles in the universe. For comparison, a semi-prime with an order of 101,000 digits corresponds to RSA's public key length of around 3,000 bits (that is becoming a standard together with 4K public keys). 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 9*, *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]*:

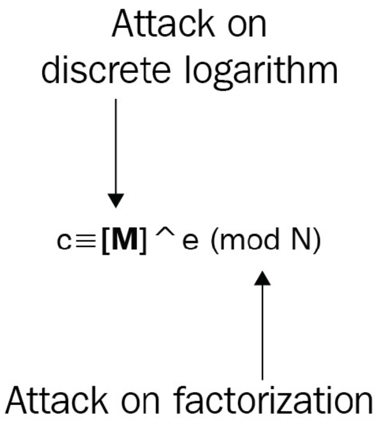


Figure 3.4: Points where it is possible to attack RSA

Most of the *conventional* analysis of cryptologists is focused on factorizing *(N)*, as we have seen. But RSA doesn't just suffer from the factorization problem; there is also another problem linked to the exponent, *(e)*. 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 Inventions in Cryptography and Logical Attacks*, 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 Inventions in Cryptography and Logical Attacks*. 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 activist in the anti-nuclear movement, 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. Even today, cryptography is still considered a military weapon. There is a license 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 MitM attack.

The following are the steps of the protocol:

1. The key is transmitted using an asymmetric encryption algorithm (ElGamal, RSA).
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 it is breakable now), Triple DES, IDEA, and AES: we covered this in *Chapter 2*, *Symmetric Encryption Algorithms*).
3. A digital signature is used to identify the users (we will look at RSA digital signatures in *Chapter 4*, *Hash Functions and Digital Signatures*).
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 MitM 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*, *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:

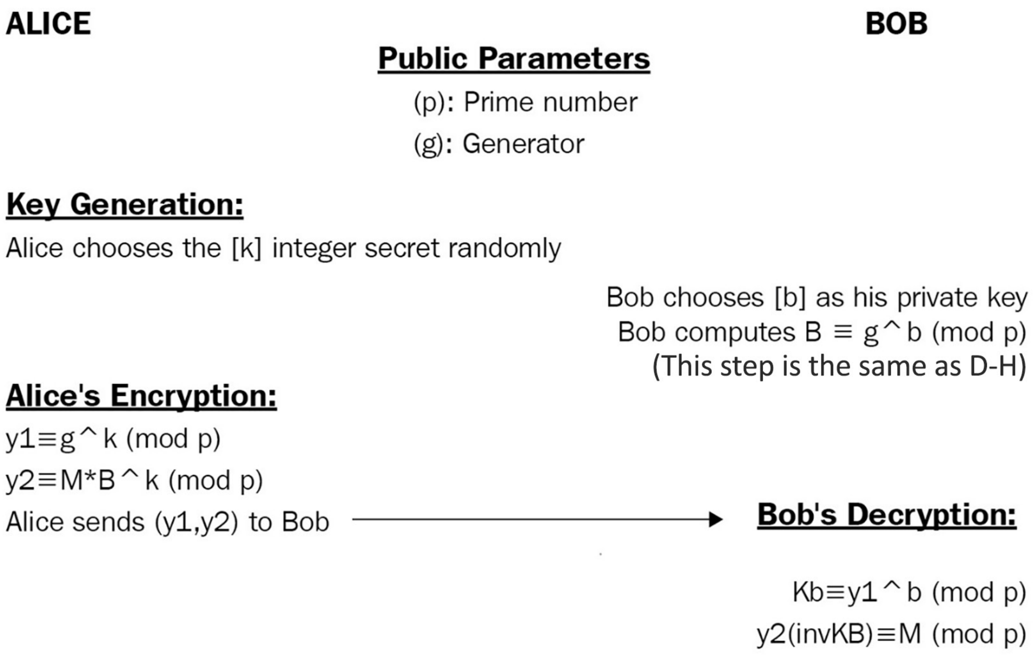


Figure 3.5: Encryption/decryption of the ElGamal algorithm

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 $A * B = 1 \pmod{p}$. 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):

- $p = 200{,}003$
- $g = 7$

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):

$$B \equiv 7^{2367} \ (mod\ 200003)$$

$$B \equiv 151854$$

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:

$$y1 \ \equiv \ 7^{23} \ (mod\ 200003)$$

$$y1 \ = \ 90914$$

$$y2 \ \equiv \ 88 * 151854^{23} \ (mod\ 200003)$$

$$y2 \ = \ 161212$$

Alice 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*:

$$Kb \equiv 90914^{2367} \ (mod\ 200004)$$

$$Kb = 10923$$

Kb is then inverted in *(mod p)* (this is *Reduce [Kb * x == 1, x, Modulus -> p]* when 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]*:

$$y2 * Inverted\ Kb \equiv M \ (mod\ 200003)$$

The final result is the message *[M]*:

$$161212 * 192331 \equiv 88 \ (mod \ 200003)$$

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.html`.

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, $B \equiv g^b \ (mod \ p)$; $Y1 \equiv g^k \ (mod \ p)$ and $Kb \equiv y1^b \ (mod \ p)$ 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)*:

$$y2 \equiv M * B^k \ (mod \ p)$$

Here, you can see that there is a multiplication. So, an attacker could also try to arrive at the message by factorizing *(y2)*.

This will be clearer if we reduce the function:

$$H \equiv B^k \ (mod \ p)$$

Then, we have the following:

$$y2 \equiv M * H \ (mod \ p)$$

This can be rewritten like so:

$$M \equiv \frac{y2}{H} (mod \ p)$$

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-Hellman, RSA, and ElGamal, work. We also explored an interesting application of RSA related to exchanging sensitive data between two nations. In *Chapter 4*, *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 Signatures

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, ensure that the document is not exposed to prying eyes.

In this chapter, we will cover the following topics:

- Hash functions
- Digital signatures with RSA and ElGamal
- 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 Encryption 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, a signature is commonly required. 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. That said, 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: one definition of hash is *to reduce into pieces*, which, in this case, refers to the contents of a message or any other information being reduced into smaller portions.

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*, *Symmetric Encryption Algorithms*, 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:

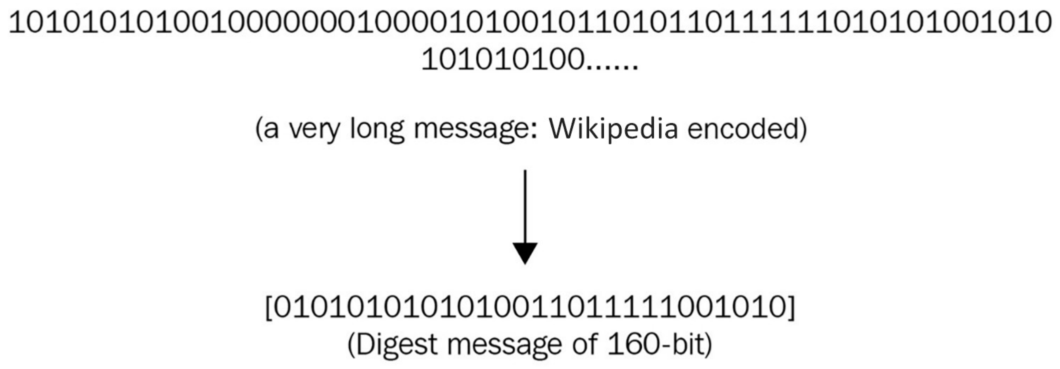


Figure 4.1: Example of hash digest

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:

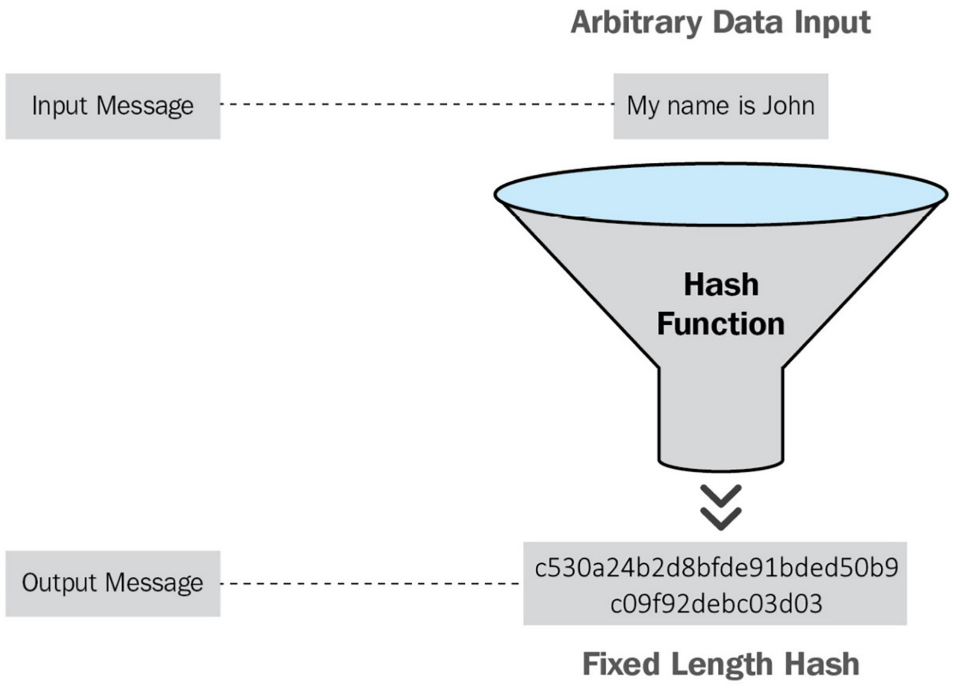


Figure 4.2: Hash functions symbolized by a "funnel mincer"

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 Wikipedia *[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 8*, *Homomorphic Encryption and Crypto Search Engine*, and will also see them in *Chapter 9*, *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:

$$g^{[a]} \equiv y \ (mod\ p)$$

As you may remember, given the output (y) and also knowing (g), it is very difficult to find the secret private key *[a]*.

However, 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.

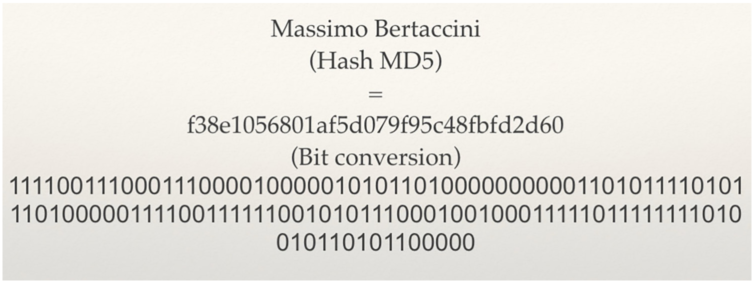
Finally, for your knowledge, there is **RIPEMD-160**, an algorithm developed by EU scientists in the early 1990s available in 160 bits and in other versions of 256 bits and 320 bits. This algorithm didn't have the same success as the SHA family, but could be the right candidate for security as it has 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. This is a free online resource, which you can find through this link: `https://www.md5hashgenerator.com/`. My name converted into hexadecimal notation with MD5 is as follows:



Massimo Bertaccini
(Hash MD5)
=
f38e1056801af5d079f95c48fbfd2d60
(Bit conversion)
11110011100011100001000000101011010000000000110101111010111010000011110011111100101011100010010001111110111111101001011010110000 0

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, Symmetric Encryption Algorithms*, and shore up some basic concepts.

Operating in Boolean logic means, as we saw in *Chapter 2*, *Symmetric Encryption Algorithms*, 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:

$$X = 60 ----> 00111100$$

$$Y = 240 ----> 11110000$$

$$AND = 48 ----> 00110000$$

- **X ∨ 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:

$$X = 60 ————> \quad 00111100$$

$$Y = 240 ————> \quad 11110000$$

$$OR = 252 ————> \quad 11111100$$

- **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:

$$a = 60 ————> \quad 00111100$$

$$b = 240 ———————> \quad \mathbf{11110000}$$

$$XOR = 204 ————> \quad 11001100$$

Other 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. This operation means to shift *(X)* bits to the left by *r* positions.

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

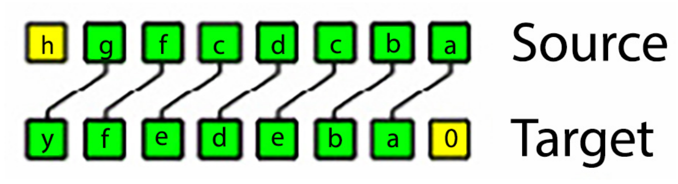


Figure 4.4: Scheme of the shift left bit operation

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

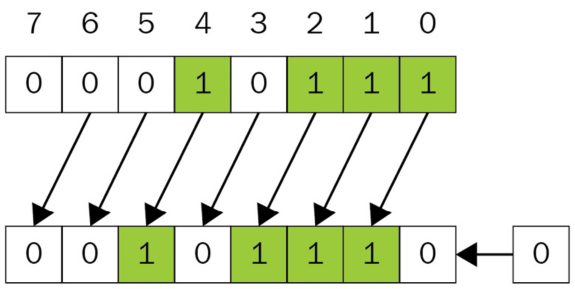


Figure 4.5: Example of the shift left bit operation (1 position)

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:

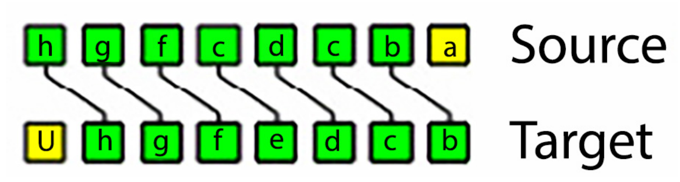


Figure 4.6: Scheme of the shift right bit operation (1 position)

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:

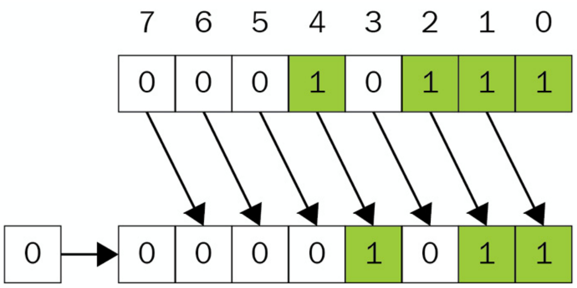


Figure 4.7: Example of the shift right bit operation (1 position)

- **X↩r**: 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.

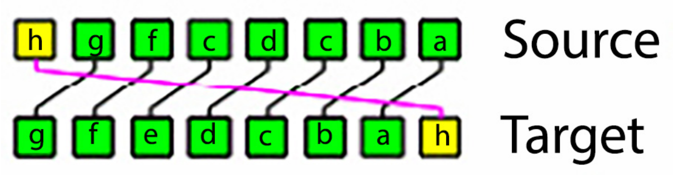


Figure 4.8: Scheme of left bit rotation (1 position)

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

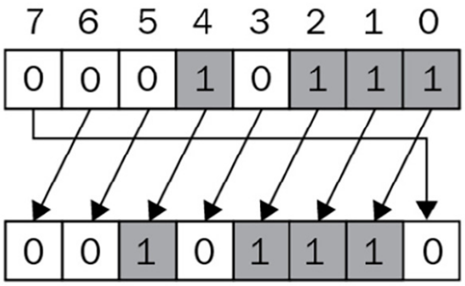


Figure 4.9: Example of left bit rotation

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

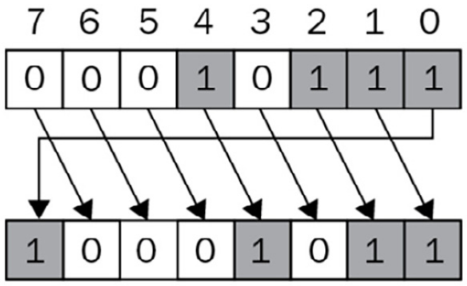


Figure 4.10: Example of right bit rotation

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

- $\boxplus$: This operator represents the modular *sum $(X+Y) \pmod{2^{32}}$* 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. I'm assuming you've heard of hex before, but if you need to, you can also take a look at *Figure 4.11*, which presents the conversion between binary, hex, and decimal numbers.

Basically, it uses the binary numbers for 0 to 9, consisting of 4 bits per number, for example:

- 0 = 0000
- 1 = 0001
- 2 = 0010

....

- 9 = 1001

Then, from 10 to 16, we have 6 capital letters, *A*, *B*, *C*, *D*, *E*, and *F*, to reach a total of 16 (hex) numbers:

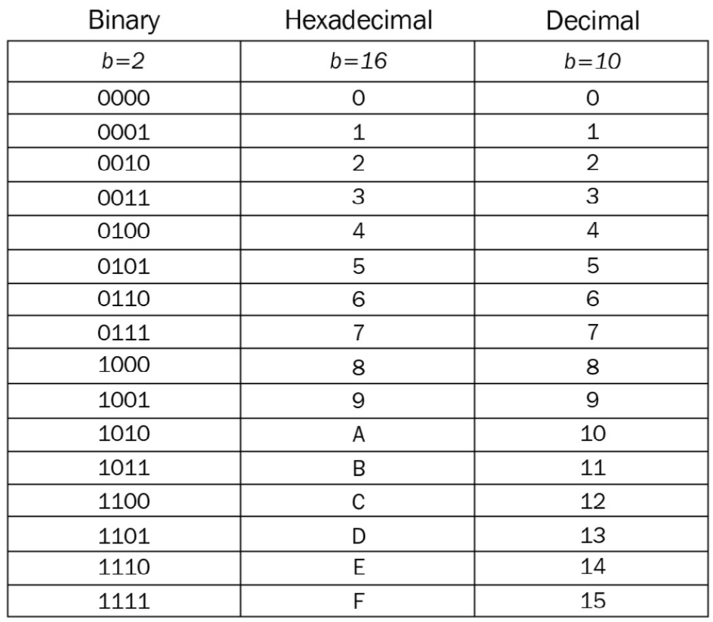
- A = 1010

- B = 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)*. We will take a closer look at this in *Step 3* of the algorithm. For now, we just need to know that it 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 the avoidance of 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:

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

Figure 4.12: The Kt constants in SHA-1

Here, 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 f_t *(B, C, D)* defined as follows:

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

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:

  $$y = m1 \parallel m2 \parallel m3 \parallel \ \dots \ \parallel mL$$

  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**: Remember how a compression function combines the current block with the result obtained in the previous round? Let's see this in practice. For *j = 0, 1, … , L-1,* execute the following instructions:

  a. $mi = W0 \parallel W1 \dots\dots\dots.. \parallel W15,$

     where each *(Wj)* consists of **32** bits.

  b. For *t = 16* to **79**, put:

     $$Wt = (Wt3 \oplus Wt8 \oplus Wt14 \oplus Wt16) \hookleftarrow 1e$$

  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 $0 \leq t \leq 79$, execute the following steps in succession:

$$T = (A \hookleftarrow 5) + ft((B, C, D) + E + Wt + Kt = D, D = C, C = (B \hookleftarrow 30), B = A, A = T$$

e. The sub-registers (A, B, C, D, E) are added to the sub-registers $(H0, H1, H2, H3, H4)$:

$$H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E$$

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

$$H0 \parallel H1 \parallel H2 \parallel H3 \parallel H4$$

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*):

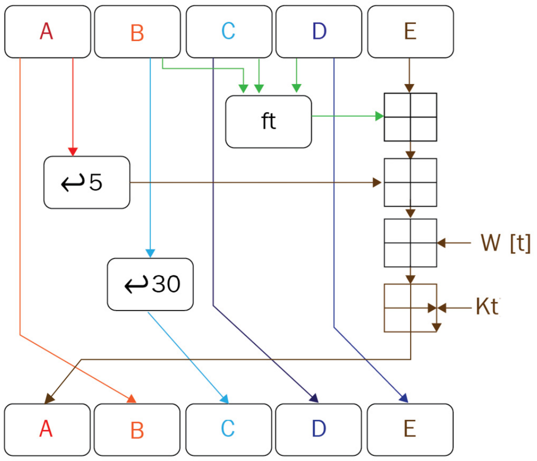


Figure 4.14: SHA-1 scheme of the operations in each sub-register

Important note

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

- $X \hookleftarrow r$ is left bit rotation, also represented as <<<, and means a circular rotation of bits. So, in the case of $A \hookleftarrow 5$, it means that bits rotate 5 positions to the left, while in the case of $B \hookleftarrow 30$ bits, bits rotate 30 positions to the left.
- ⊞: This operator represents the modular *sum* $(X+Y) \ (mod\ 2^{32})$.

As another example, we can observe the following diagram, which presents a single round of X_j:

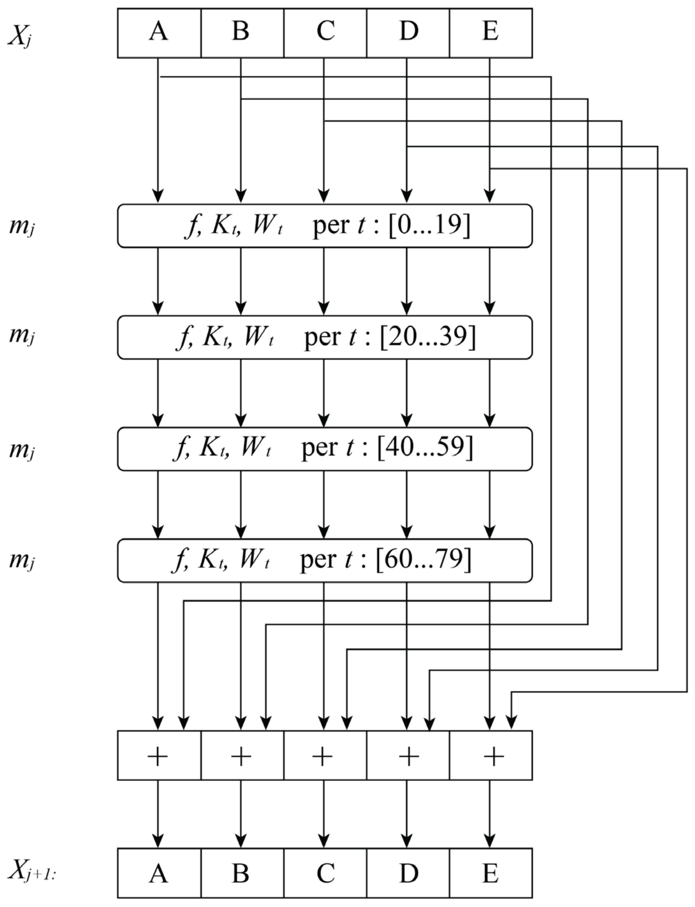

X_j
A
B
C
D
E
m_j
f, K_t, W_t per t : [0...19]
m_j
f, K_t, W_t per t : [20...39]
m_j
f, K_t, W_t per t : [40...59]
m_j
f, K_t, W_t per t : [60...79]
+
+
+
+
+
X_{j+1}:
A
B
C
D
E

Figure 4.15:

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** and **diffusion** on the bits involved in the iterative operations.

Remember that we learned about confusion and diffusion based on the Shannon Theorem in *Chapter 2*, *Symmetric Encryption Algorithms*.

As a simple recap, confusion is linked to the property of increasing the entropy of the ciphertext related to the key, augmenting the ambiguity of the ciphertext itself. Diffusion means that changing a single bit in the ciphertext (statistically) will change half of the bits in the plaintext, augmenting the difficulty of decrypting it.

The process of padding consists of the following sequence of passages:

1. SHA-1 starts by taking the original message and appends **1** bit followed by a sequence of **0** bits.
2. The **0** bits are added to ensure that the length of the new message is a multiple of **512** bits.
3. 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:

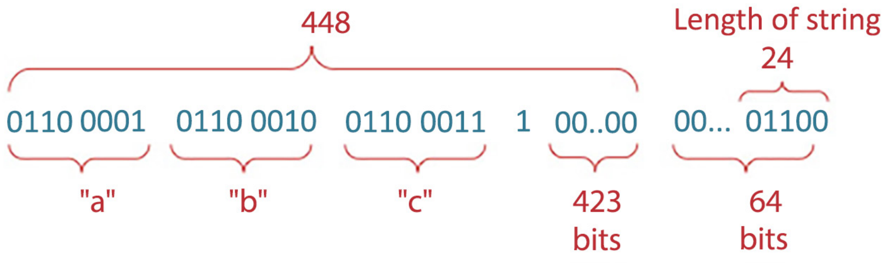


Figure 4.16: Padding of the message in SHA-1

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

  As we learned previously, the initial hash value for the sub-registers *H0, H1, H2, H3*, and *H4* will be:

  $$H[0] = 67452301$$

  $$H[1] = EFCDAB89$$

  $$H[2] = 98BADCFE$$

  $$H[3] = 10325476$$

  $$H[4] = C3D2E1F0$$

- **Step 3: Block contents:**

  W[0] = 61626380 W[1] = 00000000 W[2] = 00000000 W[3] = 00000000 W[4] = 00000000 W[5] = 00000000 W[6] = 00000000 W[7] = 00000000 W[8] = 00000000 W[9] = 00000000 W[10] = 00000000 W[11] = 00000000 W[12] = 00000000 W[13] = 00000000 W[14] = 00000000 W[15] = 00000018

  Iterations on the sub-registers:

  A B C D E

  t = 0: 0116FC33 67452301 7BF36AE2 98BADCFE 10325476

  t = 1: 8990536D 0116FC33 59D148C0 7BF36AE2 98BADCFE

  ....

  Tt = 79: 42541B35 5738D5E1 21834873 681E6DF6 D8FDF6AD

  Addition of the sub-registers:

$$H[0] = 67452301 + 42541B35 = A9993E36$$

$$H[1] = EFCDAB89 + 5738D5E1 = 4706816A$$

$$H[2] = 98BADCFE + 21834873 = BA3E2571$$

$$H[3] = 10325476 + 681E6DF6 = 7850C26C$$

$$H[4] = C3D2E1F0 + D8FDF6AD = 9CD0D89D$$

- **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

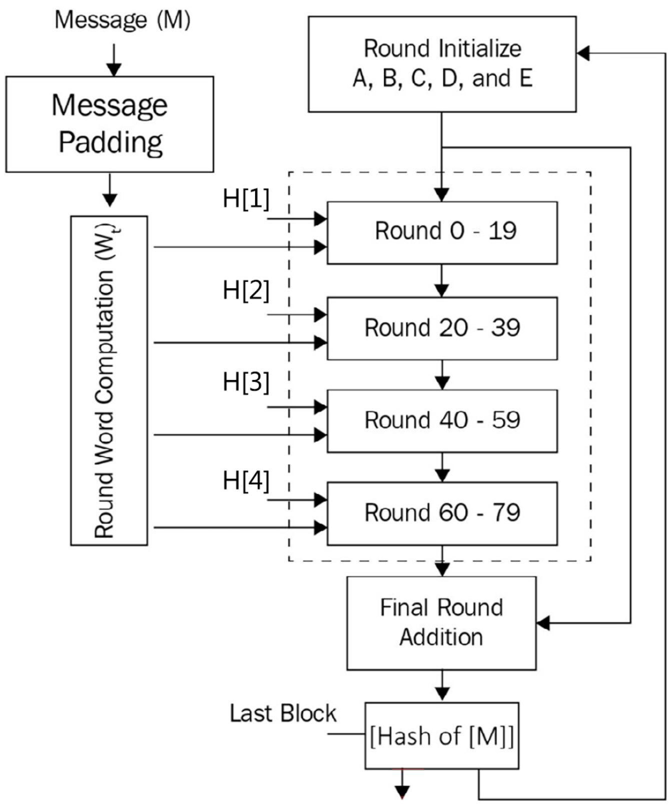


Figure 4.17: A complete round in SHA-1

Before exploring the world of digital signatures, I want to spend some time outlining the correlation between hashes and digital signatures.

Hash functions, as we will see later, are massively used in digital signatures. For example, in RSA, a hash function is required to avoid the problem of exposing the message in cleartext when the digital signature is verified by the receiver. So, it's common to substitute the message *[M]* with a hash of *[M]*.

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:

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

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*, *Zero-Knowledge Protocols*.

Let's see how a method of authentication based on a digital signature over public/private key encryption works.

Let'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:

1. How can Bob be sure that this message comes from Alice (*authentication problem*)?
2. After Bob has received the message, Alice could always deny that she is the transmitter (*repudiation*).
3. Another possible issue is that the message *[M]* could be manipulated by someone who intercepted it (a **Man-in-the-Middle** (**MitM**) attack) and subsequently, the attacker could change part of the content. This is a case of *integrity loss* of the message.
4. 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 also avoid 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 Inventions in Cryptography and Logical Attacks*, where I present some of my own algorithms and some issues found during my carrier with the relative solutions of defence.

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.

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.

Recalling the RSA algorithm from *Chapter 3*, *Asymmetric Encryption Algorithms*, 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]*:

$$[M]^e \equiv c \ (mod\ Na)$$

(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:

$$c^{[da]} \equiv [M] \ (mod\ Na)$$

where the parameter *[da]* is the private key of Alice, given by the operation:

$$\boldsymbol{INV}\ (e)\ \equiv\ [da]\ mod\ (p-1)\ (q-1)$$

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.

$$INV\ (e)\ \equiv\ [db]\ mod\ (p1-1)\ (q1-1)$$

- **Step 3**: Bob calculates his private key *[db]*
- **Step 4**: Bob performs the signature:

  $$S \equiv [M]^{[db]}\ (mod\ Nb)$$

  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:

  $$S^e \equiv [M]\ (mod\ Nb)$$

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 Algorithms*, we have the following numerical parameters given by the RSA algorithm:

- *[M] = 88*
- *e = 9007*

Alice's parameters were as follows:

- *[p] = 101*
- *[q] = 67*
- *Na = 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:

$$[M]^e \equiv c \ (mod \ N)$$

$$88^{9007} \equiv 6621 \ (mod \ 6767)$$

$$c = 6621$$

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

  Bob chooses two prime numbers, *[p1, q1]*:

$$[p1] = 211$$

$$[q1] = 113$$

$$Nb = 211 * 113 = 23843$$

$$9007 * [db] \equiv 1 \ (mod \ (211 - 1) * (113 - 1))$$

  Solving the modular equation for *[db]*, we have the following:

$$[db] = 9103$$

  Now, Bob can sign the message:

$$[M]^{[db]} \equiv S \ (mod \ Nb)$$

$$88^{9103} \equiv 19354 \ (mod \ 23843)$$

$$S = 19354$$

  Bob sends to Alice the pair *(c, S) = (6621, 19354)*.

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

$$c^{[da]} \equiv [M] \ (mod \ Na)$$

Alice calculates *[da]*:

$$9007 * [da] \equiv 1\ (mod\ (101 - 1) * (67 - 1))$$

$$[da] = 3943$$

Alice decrypts the cryptogram *(c)* and obtains the message *[M]*:

$$c^{[da]} \equiv [M]\ (mod\ Na)$$

$$6621^{3943} \equiv 88\ (mod\ 6767)$$

- **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:

$$S^{e} \equiv [M]\ (mod\ Nb)$$

$$19354^{9007} = 88\ (mod\ 23843)$$

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]*:

$$S^{e} \equiv x\ (mod\ Nb)$$

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:

$$m^{[db]} \equiv S \ (mod \ Nb)$$

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 Encryption 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')*:

$$m^{(de)} \equiv S' \ (mod \ Nb)$$

If Eve is able to trick Alice to accept the signature *(S')*, then Eve can make an MitM 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)*:

$$(S')^e \equiv m' \ (mod \ Nb)$$

$$m' \not\equiv m$$

So, Alice refuses the digital signature and will not open any message coming from this fake address. In other words, now that we have got a different result, *(m')* instead of *(m)*, Alice understands there is a problem and doesn't accept the message *[M]*.

That is why cryptographers also need to pay a lot of attention to the collisions between hashes.

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 Inventions in Cryptography and Logical Attacks*, when we explore unconventional attacks. Indeed, the exchange of public keys is highly important to prevent MitM attacks on public key infrastructure.

Digital signatures with the ElGamal algorithm

ElGamal is a public-private key algorithm, as we saw in *Chapter 3*, *Asymmetric Encryption 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. A reason why someone might prefer one over the other is that RSA is based on the factorization problem, whereas ElGamal is based on the discrete logarithm mathematical problem.

Recalling the ElGamal encryption technique, we have the following elements:

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

Alice's encryption:

$$y1 \equiv g^{[k]} \ (mod\ p)$$

$$y2 \equiv [M] * B^{[k]} \ (mod\ p)$$

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)* as follows:

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

$$[INVk] \equiv k^{(-1)}\ (mod\ p-1)$$

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

$$S\ \equiv\ [INVk] *\ (m\ -\ [a]\ * y1)\ (mod\ p-1)$$

  Alice sends to Bob the public parameters *(m, y1, S)*.

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

$$V1 \equiv\ A^{(y1)} * y1\text{^}(S)\ (mod\ p)$$

After the decryption step, if *h[M] = m*, Bob obtains the second parameter of verification, *V2*:

$$V2 \equiv g^m\ (mod\ p)$$

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 = 7*
- *p = 200003*
- *h[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 Alice)*
- *[k] = 23 (random secret number of Alice)*
- *B = 151854 (public key of Bob)*
- *A = 43725 (public key of Alice)*
- *y1 ≡ g[k] (mod p) = 723 (mod 200003) = 90914*

After 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)*:

$$[INVk] \equiv [k]^{(-1)}\ (mod\ p-1) = 23^{-1}\ (mod\ 200003\ -\ 1) = 34783$$

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

$$S\ \equiv\ [INVk]\ *\ (m\ -\ [a]\ * y1)\ (mod\ p-1)$$

$$S\ \equiv\ 34783\ *\ (77{:}\ 5433\ *\ 90914)\ (mod\ 200003\ -\ 1)$$

$$S\ =\ 72577$$

  Alice 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)*:

$$V1 \equiv A^{(y1)} * y1^{(S)}\ (mod\ p)$$

$$V1 \equiv 43725^{(90914)} * 90914^{72577}\ (mod\ 200003) = 76561$$

$$V1 = 76561$$

  Bob's verification *(V2)*:

$$V2 \equiv g^{m}\ (mod\ p)$$

$$V2 \equiv 7^{77}\ (mod\ 200003) = 76561$$

$$V2 = 76561$$

Bob 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:

$$V1 \equiv A^{(y1)} * y1^{(S)}\ (mod\ p)$$

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 Encryption Algorithms*, the following function could also be an issue:

$$y2 \equiv [M] \ * \ B^{[k]} \ (mod\ p)$$

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 Inventions in Cryptography and Logical Attacks*, 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 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:

$$INV\ (e) \equiv [da]\ mod\ (pa-1)\ (qa-1)$$

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:

  $$t \equiv [M1]\ *\ [k]^e\ (mod\ Na)$$

  Bob sends *(t)* to Alice.

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

  $$S\ \equiv\ t^{[da]}\ (mod\ Na)$$

  Alice sends *(S)* to Bob, who can verify whether the blind signature corresponds to the message *[M1]*.

- **Step 3**: Verification:

  Bob calculates *(V)*:

$$S/k \equiv V\ (mod\ Na)$$

  Then he can verify the following:

$$V^e \equiv [M1]\ (mod\ Na)$$

If the last operation is *TRUE*, it means Alice has effectively signed *blind [M1]*. In this case, Bob can be sure of the following:

$$[M1]^{[da]} \equiv V\ (mod\ Na)$$

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 Encryption 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 9*, *Quantum Cryptography*.

Let's see an example to better understand the protocol.

Numerical example:

1. The parameters defined by Alice are as follows:

   - pa = 67
   - qa = 101

   So, the public key *(Na)* is the following:

$$67 * 101 = Na = 6767$$

$$da\ \equiv\ 1/e\ (mod\ (pa-1)*(qa-1))$$

$$Reduce\ [e * x\ ==\ 1, x, Modulus -> \ (pa\ -\ 1) * (qa\ -\ 1)]$$

$$[da] = 1553$$

$$e = 17$$

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

    - k = 29

    Bob calculates *(t)*:

$$t \equiv M1 * \ k^e \equiv 88 * 29^{17} = 3524 \ (mod \ 6767)$$

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

$$S \equiv t^{[da]} \equiv 3524^{1553} = 1533 \ (mod \ 6767)$$

4. Bob can verify *(S)* <——————————— Alice sends *(S = 1553)* to Bob:

$$S/k \equiv V \ (mod \ Na)$$

$$1533/29 = 2853 \ (mod \ 6767)$$

$$Reduce \ [k * x \ == \ S, x, Modulus -> \ Na]$$

$$x = 2853$$

$$V = 2853$$

If:

$$V^e \equiv [M1] \ (mod \ Na)$$

$$2853^{17} \equiv 88 \ (mod \ 6767)_,$$

then 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 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 perform a double-check on *[M1]*, so you will be able to realize that Alice has really signed *[M1]* without knowing anything about its value:

$$M1^{[da]} \equiv V \ (mod \ Na)$$

$$88^{1553} \equiv 2853 \ (mod \ 6767)$$

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 adopting such protocols.

Another consideration concerns possible attacks.

As you see here, we are faced with a factorization problem:

$$t \equiv M1 * [k]^e \ (mod\ Na)$$

We can see that *(t)* is the product of *[X*Y]*:

$$X = M1 \ <\!\!- \ Factorization\ problem$$

$$Y = k^e$$

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*.

On 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 *Chapter 5*, *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*, *Zero-Knowledge Protocols*, and the other algorithms discussed in *Chapter 6*, *New Inventions in Cryptography and Logical Attacks*. 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.

Section 3

New Cryptography Algorithms and Protocols

In this section, we will describe the new protocols and algorithms that are emerging for data protection in the new environment of cybersecurity, blockchain, ICT, and crypto search engines. Among these will be some patents and inventions from myself.

This section comprises the following chapters:

- *Chapter 5*, *Zero-Knowledge Protocols*
- *Chapter 6*, *New Inventions in Cryptography and Logical Attacks*
- *Chapter 7*, *Elliptic Curves*
- *Chapter 8*, *Homomorphic Encryption and Crypto Search Engine*

5

Zero-Knowledge Protocols

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.

Hackers know everything about their victims' lives – finances, assets owned, and even credit card numbers. 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 your name, surname, address, social security number, or PIN, but this is not always possible. Another way is to avoid exposing private information by giving **proof of knowledge**.

Proof of knowledge is a way to prove that you know something *without revealing that information*. In practice, proof of knowledge allows the user to identify themselves without entering a PIN, a password, or any other sensitive information. 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:

- The framework and logical basis of Zero-Knowledge Protocols
- Non-interactive and interactive ZKPs (the Schnorr protocol), with examples and possible attacks on them
- One-round ZKPs
- Introduction to zk-SNARKs
- 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*, a 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)**.

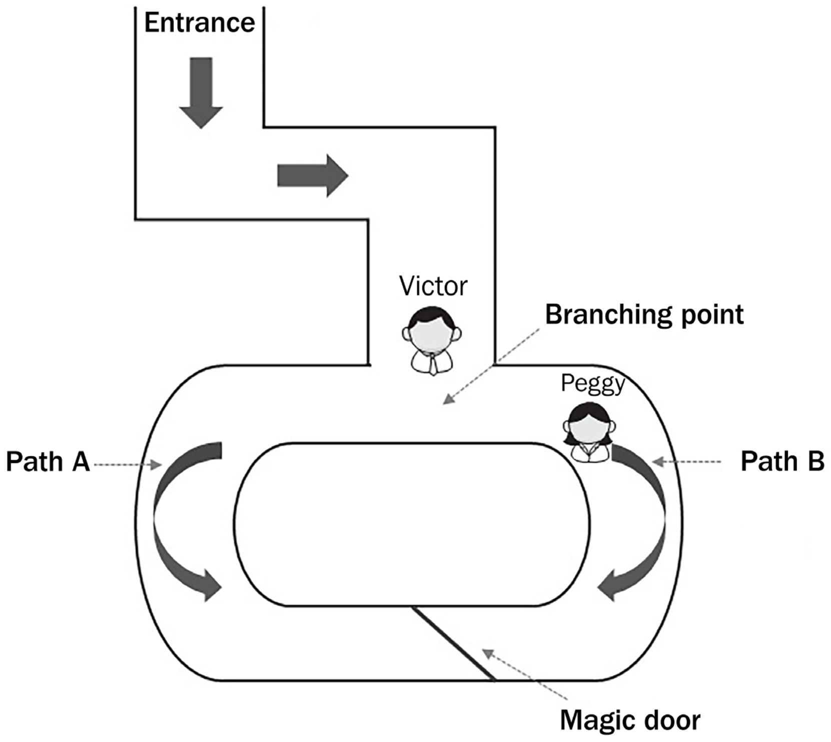


Figure 5.1: Ali Baba's cave

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, as 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 8*, *Homomorphic Encryption and Crypto Search Engine,* where we will use ZKPs to protect against man-in-the-middle attacks.

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 *[m],* encrypted with RSA, as follows:

$$m^e \equiv c\ (mod\ N)$$

Keep in mind that Peggy doesn't want to reveal the content *[m]* to Victor.

Here, *(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:

1. Peggy chooses a random integer, *[r_1]* (which she keeps secret), and calculates the inverse of r_1 (represented as *INV[r_1]*) multiplied by *[m]* (modulo *N*):

   $$r_2 \equiv [m] * r_1^{-1}\ (mod\ N)$$

   Peggy keeps r_2 secret.

2. Peggy calculates (x_1) and (x_2), as follows:

   $$x_1 \equiv r_1^{\ e}\ (mod\ N)$$

   $$x_2 \equiv r_2^{\ e}\ (mod\ N)$$

   Peggy sends x_1 and x_2 to Victor.

3. Finally, Victor verifies the following:

   $$x_1 * x_2 \equiv c\ (mod\ N)$$

It is supposed here that if Victor can verify *step 3*, $x_1 * x_2\ \equiv\ c\ (mod\ N)$, then Peggy really should know *[m]*.

As you can 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 is irrelevant if Victor actually knows *[m]* or not.

So the underlying challenge implicit in 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 $x_1 * x_2 \equiv \ c \ (mod \ N)$; otherwise, she will not be able to do that.

Another important consideration is the following: if (*c*) is a big number, it is supposed that it should be hard to know x_1 and x_2 (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, (r_2) is calculated using *[m]* in the following equation:

$$r_2 \equiv [m] * r_1{}^{-1} \ (mod \ N)$$

So the final effect of the multiplication between $x_1 * x_2$ (as you can see in the following demonstration) will be to eliminate (r_1) 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 Algorithms*); here, the function is as follows:

$$x_1 * x_2 \equiv c \ (mod \ N)$$

This states that it is computationally hard to find two numbers (x_1, x_2) 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.

Let's see with a numerical example how this protocol works before going deeper to analyze it:

- *m = 88*
- *N = 2430101*
- *e = 9007*
- *me ≡ c (mod N)*
- *889007 ≡ 160613 (mod 2430101)*

Now, let's start the protocol of verification.

Peggy chooses a random number:

$$r_1 = 67$$

- **Step 1**: Peggy calculates r_2:

  $$r_2 \equiv [m] * {r_1}^{-1} \ (mod\ N)$$

  First, Peggy calculates the *[Inv(r_1)]* function (the inverse value of r_1 *(mod N)*), and then she multiplies it by *[m]:*

  $$67 * x \equiv 1 \ (mod\ 2430101)$$

  $$x = 217621$$

  $$[m] * x \equiv r_2 \ (mod\ N)$$

  $$88 * 217621 \equiv 2139941 \ (mod\ 2430101)$$

  So Peggy gets r_2:

  $$r_2 = 2139941$$

  Then, Peggy calculates x_1 and x_2:

  $$x_1 \equiv {r_1}^e \ (mod\ N)$$

  $$x_2 \equiv {r_2}^e \ (mod\ N)$$

  $$67^{9007} = 1587671 \ (mod\ 2430101)$$

  $$x_1 = 1587671$$

  $$2139941^{9007} \equiv 374578 \ (mod\ 2430101)$$

  $$x_2 = 374578$$

  Finally, Peggy sends x_1, x_2 *(1587671, 374578)* to Victor.

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

  $$x_1 * x_2 \equiv c \ (mod\ N)$$

  $$1587671 * 374578 \equiv 160613 \ (mod\ 2430101)$$

  $$160613 = c$$

Let's see why this protocol is mathematically correct.

Demonstration of a non-interactive ZKP

We have to demonstrate the following:

$$x_1 * x_2 \equiv c \ (mod\ N)$$

Where $c \equiv [m]^e \ (mod\ N)$, we can substitute in the previous function $(x_1 = r_1{}^e)$ with $(x_2 = r_2{}^e)$ so that we get the following:

$$x_1 * x_2 \equiv r_1{}^e * r_2{}^e \equiv c \ (mod\ N)$$

Substituting r_2 into the equation, we have the following:

$$x_1 * x_2 \equiv (r_1)^e * (\boldsymbol{m} * (r_1{}^{-1})^e) \equiv (r_1)^{\boldsymbol{e}} * m^e * (\boldsymbol{Inv}(r_1)^{\boldsymbol{e}}) \equiv c \ (mod\ N)$$

Going by the modular power's properties (collecting together the two factors I have highlighted), we have the following:

$$r_1{}^e * \left(Inv(r_1)\right)^e \equiv 1 \ (mod\ N)$$

So eliminating $r_1{}^e$ and $(Inv(r_1))^e$ 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:

$$x_1 * x_2 \equiv m^e \equiv c \ (mod\ N)$$

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 $x_1 * x_2 = c$. That's just what we wanted to demonstrate.

This protocol has an important characteristic – it's executed in only two steps:

1. Peggy calculates the parameters
2. Victor verifies the correctness

Crucially, this avoids any interaction between Peggy and Victor.

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 that 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. Over time, we have since seen more demonstratable attacks on a ZKP.

The goal of this attack is to demonstrate that Eve (the attacker) can calculate two *fake* numbers (x_1, x_2) that prove to factorize (c), even if Eve effectively doesn't know *[m]*.

Let's explore how this attack works and what effects are produced:

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

$$[r] * (v1) \equiv e \ (mod\ N)$$

   (e) is the public parameter of RSA (as we saw in the *RSA* section of *Chapter 3*, *Asymmetric Encryption Algorithms*), so *(e)* is known by everyone. By means of this function, Eve can extract *(v1)*.

2. In parallel, Eve calculates the following:

$$e * x \equiv c \ (mod\ N)$$

3. The parameters *(e, c)* are 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*):

$$x * r \equiv v2 \ (mod\ N)$$

   Eve sends (*v1*, *v2*) to Victor, who can verify the following:

$$v1 * v2 \equiv c \ (mod\ N)$$

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*:

$$e * r^{-1} \equiv v1 \ (mod\ N)$$

$$9007 * 436172 = 1557988 \ (mod\ 2430101)$$

$$v1 = 1557988$$

  Eve performs *v2*.

The next operation is to obtain the inverse of (e) with respect to (c).

$e * x \equiv c\ (mod\ N)$, obtaining (x) in inverse modular multiplication:

$$9007 * x = 160613\ (mod\ 2430101)$$

$$x = \mathbf{2031892}$$

Then, using **x**, Eve gets *v2*, performing the following operation:

x * r ≡ v2 (mod N), obtaining v2:

$$v2 = \mathbf{1480556}$$

After having gained *v2*, Eve sends (*v1 = 1557988*; *v2 = 1480556*) to Victor.

- **Step 2:** The 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 that this is *not* an authentication protocol; it's a proof of statement (or knowledge) that Peggy knows *[m]*.

For completeness, also note that (c) is not a product of two big prime numbers such as N is, as you have probably already noticed in the example. So the logical basis of this protocol is weak: finding two numbers whose product is (c) would be not so difficult, even if (c) is a large number.

To use a hypothetical example, imagine 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 prevented under certain conditions, one of which 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 (x_1 and x_2) 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 prevent problems such as this, 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 shows Victor that she knows 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).
- $B \equiv g^a \ (mod\ p)$ 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]* is 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 $1 \leq k < p\text{-}1$. This is the basis of modular math.

She performs the following calculation:

$$V \equiv g^k \ (mod\ p)$$

Peggy sends (V) to Victor.

- **Step 2**: Victor chooses a random integer, (r), where $1 \leq r < p\text{-}1$.

  Victor sends (r) to Peggy.

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

$$w \equiv (k - a * r)(mod\ p - 1)$$

  Peggy sends (w) to Victor.

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

$$V \equiv g^w * B^r \ (mod\ p)$$

  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:

$$V \equiv g^{\boldsymbol{k}} \ (mod\ p)$$

$$B \equiv g^{\boldsymbol{a}} \ (mod\ p)$$

$$w \equiv k - a * r \ (mod\ p - 1)$$

Now, we substitute all the past equations inside the last verification of *step 3*:

$$V \equiv g^w * B^r \ (mod\ p)$$

$$V \equiv g^k \ (mod\ p)$$

Substituting the functions in (**V**), the equation becomes the following:

$$V \equiv \left(g^{(k-a*r(mod\ p-1))}\right) * ((g^a)^r)(mod\ p)$$

For the properties of exponential factors, we have the following:

$$g^k \equiv g^{(k-[ar])} * g^{[ar]} \ (mod\ p)$$

$$V \equiv g^k \equiv g^{(k-ar+ar)}$$

Simplifying *[-ar]* with *[+ar]*, we get back the following:

$$g^k \equiv g^k \ (mod\ p)$$

That's what we wanted to demonstrate.

Let's do a numerical example to better visualize how this interactive ZKP works.

Numerical example:

Let's assume the following parameters:

- p = 1987
- a = 17
- g = 3

(*p = 1987* and *g = 3*) are public parameters.

[a] = 17 is the secret number that Peggy claims to know:

$$B \equiv g^a \ (mod\ p)$$

(*B*) is the public key of Peggy, given by the following:

$$3^{17} \equiv 1059 \ (mod\ 1987)$$

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 the following:

$$k - a * r \equiv w \ (mod\ p - 1)$$

$$67 - 17 * 37 \equiv 1424 \ (mod\ 1987 - 1)$$

$$w = 1424$$

Peggy sends (*w = 1424*).

Finally, Victor now verifies whether *(V) = 1753* corresponds to the following:

$$g^w * B^r \equiv V\ (mod\ p)$$

$$3^{1424} * 1059^{37} \equiv 1753\ (mod\ 1984)$$

$$V = 1753$$

In 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 saw in *Chapter 3*, *Asymmetric Encryption Algorithms*, knowing *(B)* and *(g)* is not enough to compute *[a]* in this function:

$$B \equiv g^{[a]}\ (mod\ p)$$

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)*:

$$w \equiv k - \boldsymbol{a} * r\ (mod\ p - 1)$$

There is no way to trick Victor, who, furthermore, sent (*r*) to Peggy, which is used in the last verification function together with (*v*) and (*B*), along with (*r*), to ensure that Peggy cannot bluff:

$$V \equiv g^w * \boldsymbol{B^r}\ (mod\ p)$$

So I hope I have convinced you that there is no way for Peggy to trick Victor in this case.

A challenge for a disruptive attack on an interactive ZKP

Now that we have seen that Peggy can't trick Victor, let's consider an attack against this protocol that I created in late 2018 and 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 $1 \le k < p-1$.

  Then, she calculates the following:

  $$V \equiv g^{[k]}\ (mod\ p)$$

  It's when Peggy sends (*V*) to Victor that Eve can try to launch a man-in-the-middle attack.

  Peggy sends (*V*) to Victor.

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

  Eve injects $V1 \equiv g^{[k1]}\ (mod\ p)$, 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:

  $$V1 \equiv g^{(\boldsymbol{v1})} * \boldsymbol{B^r}\ (mod\ p)$$

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 things to note:

- 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 person is *ON*, the other person is probably *OFF* because they are sleeping. In that case, this isn't probably the most appropriate protocol to use.

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.

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, from the University of Southern California. It purports to give proof of knowledge for a challenge in just one round. The paper, *Security and privacy using one-round zero-knowledge proofs* (2005), 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:

$$g^{[x]} \equiv b \ (mod\ p)$$

Victor launches a challenge (c) to verify whether Peggy really knows *[x]*. He picks up a random *[y]* and calculates the following function:

$$c \equiv g^{[y]}\ (mod\ p)$$

Victor sends (c) to Peggy. She inserts the parameter *[x]* on (c), computing (r) as follows:

$$c^{[x]} \equiv r\ (mod\ p)$$

Peggy sends (r) to Victor, who can verify the following:

$$r \equiv b^{[y]}\ (mod\ p)$$

Finally, Victor accepts the verification if *(r)* corresponds to $V = b^{[y]}\ (mod\ p)$.

This protocol looks very simple and straightforward; you may recall some of it from the **Diffie-Hellman (D-H) algorithm** we covered in *Chapter 3*, *Asymmetric Encryption Algorithms*. It is based on the computational difficulty of calculating 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:

$$\boldsymbol{r} \equiv c^{[x]} \equiv {g^{[y]}}^{[x]} \equiv \boldsymbol{g}^{[\boldsymbol{y}*\boldsymbol{x}]}\ (mod\ p)$$

$$V \equiv b^{[y]} \equiv {g^{[x]}}^{[y]} \equiv \boldsymbol{g}^{[\boldsymbol{x}*\boldsymbol{y}]}\ (mod\ p)$$

As you can see, $r \equiv V \equiv b^{[y]}\ (mod\ p)$.

Numerical example

Let's look at a numerical example:

- $p = 2741$
- $g = 7$

$x = 88$ is the secret number that Peggy has to demonstrate she knows.

The statement is as follows:

$$g^x \equiv b\ (mod\ p)$$

$$7^{88} \equiv 1095\ (mod\ 2741)$$

$$b = 1095$$

Victor chooses a random number:

$$y = 67$$

Victor calculates the following:

$$g^y \equiv c\ (mod\ p)$$

$$7^{67} \equiv 1298\ (mod\ 2741)$$

$$c = \mathbf{1298}$$

Peggy, after having received (c), calculates the following:

$$c^x \equiv r\ (mod\ p)$$

$$1298^{88} \equiv 361\ (mod\ 2741)$$

$$r = \mathbf{361}$$

Peggy sends (r) to Victor, who can verify the following:

$$b^y \equiv V\ (mod\ p)$$

$$1095^{67} \equiv 361\ (mod\ 2741)$$

$$\boldsymbol{V} = \mathbf{361} = \boldsymbol{r}$$

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, we will examine some notes that further demonstrate the strong similarity of this protocol with the D-H algorithm.

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:

  $$g^x \equiv b \ (mod\ p)$$

  That is the same in D-H, as follows:

  $$g^a \equiv A \ (mod\ p)$$

- **Step 2**: Victor:

  $$g^y \equiv c \ (mod\ p)$$

  That is the same in D-H, as follows:

  $$g^b \equiv B \ (mod\ p)$$

- **Step 3**: Peggy:

  $$c^x \equiv r \ (mod\ p)$$

  In D-H, this becomes the shared key H:

  $$B^a \equiv H \ (mod\ p)$$

- **Step 4**: Victor:

  $$b^y \equiv r \ (mod\ p)$$

  In D-H, this again becomes the shared key H:

  $$A^b \equiv H \ (mod\ p)$$

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 Algorithms*.

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:

$$c \equiv g^y \ (mod\ p)$$

Step 2: Peggy sends the following to Victor:

$$r \equiv c^x \ (mod\ p)$$

Instantly, Victor can verify the following:

$$r \equiv b^y \ (mod\ p)$$

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 inspired me 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.

An introduction to zk-SNARKs – spooky moon math

Now, we are approaching a new kind of protocol called **zk-SNARKs** – a kind of non-interactive ZKP that is a little bit complicated, which is also known as **spooky moon math**. 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 types 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 identify users in cryptocurrency schemes, as we will see in this section.

zk-SNARKs have been adopted in the blockchain as a scope of authentication to create *consensus*.

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 navigation in 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 proof of knowledge.
- There is no mathematical way to trick the verifier with a shortcut or fake proof (such as substituting fake parameters in the $V1 \equiv g^{(v1)} * B^r \ (mod\ p)$ equation in order to avoid knowing *[a]*).

Now, 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 public key, (y), given by the following:

$$y \equiv g^a \ (mod\ p)$$

(g) is a generator (as in D-H and 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:

$$t \equiv g^v \ (mod\ p)$$

Anna calculates (*c*) as a hash function of the three parameters, (*g, y, t*), and she can compute (*r*) as follows:

$$r \equiv v - c * a \ (mod\ p-1)$$

The verifier, Carl, can check the following:

$$t \equiv g^r * y^c \ (mod\ p)$$

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. In practice, this attack may face issues encountering protected information, so the purpose is instead to show that it would be mathematically possible.

Let's say that Eve is a server. We suppose that Eve intercepts the *H(g, y, t)* public hash function and performs a man-in-the-middle 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:

$$t1 \ \equiv \ g^{v1} \ (mod\ p)$$

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 insert (r_1) as follows:

$$r_1 \ = \ v1$$

Now, Eve orders the third server connected to the internet to send to Carl (r_1, *v1, c1*), who can verify the following:

$$t1 \equiv g^{r_1} * y^{c1} \ (mod\ p)$$

It's simple to demonstrate that $t1 = g^{v1}$ because of the following:

$$y^{(p-1)} \equiv 1 \ (mod\ p)$$

Finally, as we have assigned $c1 = p - 1$ and $r_1 = v1$, the final effect will be as follows:

$$t1 \equiv g^{v1} \equiv g^{v1} * 1 \ (mod\ p)$$

Numerical example:

- *p = 3571*
- *g = 7*
- *x = 23*

Anna's public key is as follows:

$$y \equiv g^{x} \ (mod\ p)$$

$$7^{23} = 907 \ (mod\ 3571)$$

$$y = \mathbf{907}$$

Now, I will show you how Anna can demonstrate to Carl how to get the secret number, *[x]*.

She chooses *v = 67*, as follows:

$$t \equiv g^{v} \ (mod\ p)$$

$$7^{67} = 584 \ (mod\ 3571)$$

$$t = \mathbf{584}$$

Let's suppose that hash (*g, y, t*) is as follows:

$$c = \mathbf{37}$$

She computes *r* as follows:

$$r \equiv v - c * x \ (mod\ p - 1)$$

$$(67 - 37 * 23) \equiv 2786 (mod\ 3570 - 1)$$

$$r = \mathbf{2786}$$

Anna sends $(r, t, c) = (2786, 584, 37)$ to Carl.

Carl can verify the following:

$$g^r * y^c \equiv t\ (mod\ p-1)$$

$$7^{2786} * 907^{37} = 584\ (mod\ 3571)$$

However, Eve intercepts the public parameters (*y*), (*t*), and (*r*). Eve leaves the (*y*) invariant, but she changes (*t*) to (*t1*) and (*r*) to (r_1), performing a man-in-the-middle attack:

$$v1 = 57$$

Eve calculates the following:

$$t1 \equiv 7^{57}\ (mod\ 3571)$$

$$\boldsymbol{t1 = 712}$$

$$r_1 = v1 = 57$$

$$c1 = p-1 = 3570$$

Eve sends $(r_1, t1, c1) = (57, 712, 3570)$ to Carl.

Carl verifies the following:

$$\boldsymbol{t1} \equiv g^{r_1} * y^{c1}\ (mod\ p)$$

I highlighted the parameter that Eve substitutes, *(t1, r_1, c1)*; she left the (*y, g, p*) invariant.

Substituting the new parameters into the equation of verification, we have the following:

$$7^{57} * 907^{3570} \equiv 712\ (mod\ 3571)$$

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 $r_1 = v1$ 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 prevent it.

Note that the parameter (*y*), the public key from 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.

ZK13 – a ZKP for authentication and key exchange

Now, we will look at the **ZK13** protocol, which I patented in 2013. It's a non-interactive protocol that solves an issue that is very important to the **Crypto Search Engine** (**CSE**) project explained in *Chapter 8*, *Homomorphic Encryption and Crypto Search Engine*. Essentially, the issue is this: we need authentication without requiring a public key.

In this section, we will analyze the ZKP that's used for authentication; 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 there are several problems that this ZKP can solve. 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. We will talk in detail about the CSE in *Chapter 8*, *Homomorphic Encryption and 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 a **Virtual Machine** (**VM**) 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, Symmetric Encryption Algorithms*, 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. However, 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 prevent 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 prevent 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:

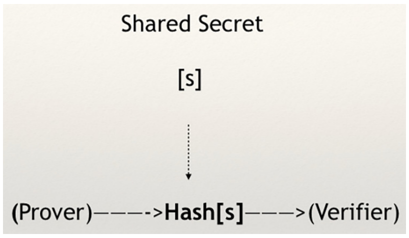


Figure 5.2: The scheme of the shared hash[s] secret

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:

$$r \equiv g^k \ (mod\ p)$$

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):

$$\boldsymbol{H}[\boldsymbol{s}] * \boldsymbol{k} \equiv \boldsymbol{F}\ (mod\ p-1)$$

Now, with (*g*) raised to *[F]*, Bob proves (*P*), which is the second element of the zk-SNARK:

$$g^F \equiv P\ (mod\ p)$$

Bob sends the pair (*P, r*) to Alice.

Step 2: The verification step (*V*) validates the proof, (*P*), based on the function:

$$[s] \text{——>}\ H[s]$$

$$r^{[Hs]} \equiv g^F\ (mod\ p)$$

If:

$$V \equiv r^{[Hs]} = P\ (mod\ p)$$

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 = 2741
- g = 7

Secret parameters:

- H[s] = 88
- k = 23
- $g^k \equiv r$ (mod p)
- $7^{23} \equiv 2379$ (mod 2741)
- r = 2379

Now, Bob calculates *[F]* and then (*P*):

$$[Hs] \ * \ k \ \equiv \ F \ (mod\ p-1)$$

$$88 \ * \ 23 \ \equiv \ 2024 \ (mod\ 2741-1)$$

$$F \ = \ \mathbf{2024}$$

$$g^{F} \ \equiv \ P \ (mod\ 2741)$$

$$7^{2024} \ \equiv \ 132 \ (mod\ 2741)$$

$$P \ = \ \mathbf{132}$$

Alice verifies the following:

$$r^{\ [Hs]} \equiv \ P \ (mod\ p)$$

$$2379^{88} \ \equiv \ 132 \ (mod\ 2741)$$

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 $P \ \equiv \ g^{F} \ (mod\ p)$, what we want to demonstrate is the following:

$$P \ \equiv \ g^{F} \ \equiv \ r^{S} \ (mod\ p)$$

(Here, I use *[s]* for the demonstration instead of *H[s]*.)

As $r \ \equiv \ g^{k} \ (mod\ p)$, substituting (*r*) in the preceding equation, we have the following:

$$P \ \equiv \ g^{F} \ \equiv \ (g^{\wedge k})^{\wedge s} \ (mod\ p)$$

We also know that *F* is the following:

$$F \ \equiv \ s * k \ (mod\ p-1)$$

Finally, for the properties of the modular powers substituting both *[F]* and (*r*), we get the following:

$$P \ \equiv \ g^{s*k} \ \equiv \ (g^{\wedge k})^{\wedge s} \ \equiv \ g^{\wedge k*s} \ (mod\ p)$$

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 by 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 divided it into 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 (r_1), generating a fake (*k1*), by calculating the following:

$$r_1 \equiv g^{k1} \ (mod\ p)$$

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 prevent 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 proposed an attack attempt.

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.

Later in the chapter, we encountered and analyzed a non-interactive protocol based on the D-H algorithm. We explored ZK13, a non-interactive protocol, and its use of shared secrets to enable the authentication of VMs.

Finally, we explored zk-SNARKs in the world of cryptocurrency, especially those used to anonymize transactions.

You became familiar with some attack schemes, 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 topics that refer 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.

Join us on Discord!

Read this book alongside other users. Ask questions, provide solutions to other readers, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/SecNet`

6

New Inventions in Cryptography and Logical Attacks

In *Chapter 5*, *Zero-Knowledge Protocols*, I already explained my non-interactive protocol that was patented in 2013: ZK13. In this chapter, I will explain two algorithms that were patented and published between 2008 and 2012. The two algorithms we will talk about are called MB09 and MBXI: the acronyms formed of my initials combined with the year of their invention. Additionally, we will look at a third protocol, MBXX, patented in 2020, which is a union between MB09 and MBXI from a new perspective that is related to blockchain and digital currency. Finally, I will introduce Lightweight encryption, a new algorithm for encrypting data for IoT and smart devices.

We will begin with an introduction to the genesis of these algorithms. Following this, we will examine, in detail, the schemes of the algorithms, their strengths, and their weak points.

We will start with MB09, which is a public/private key algorithm concept used principally for digital money transmission. Moreover, we will see the digital signature schemes of MBXI and a comparison with Diffie-Hellmann (D-H) and RSA.

Finally, we will discuss one of the last cryptographic protocols, MBXX, that could be considered valid for blockchain consensus.

In this chapter, we will cover the following topics:

- The genesis of the MB09 algorithm
- The scheme and explanation of MB09
- A detailed explanation of MBXI
- Unconventional attacks on RSA and public key encryption
- Digital signatures in MBXI
- MBXX: the evolution of MB09 and MBXI in the light of the blockchain revolution and the consensus problem
- Lightweight encryption and the Cybpher algorithm

Several of the algorithms we will look at are related to the *digital payment* environment, so we will dedicate a part of this chapter to discussing this environment, touching on blockchain and cryptocurrency.

Now, let's look at the genesis of the first algorithm, MB09, and its related applications.

The genesis of the MB09 algorithm and blockchain

When I started project MB09 and, consequently, MBXI, at the time, I didn't know which applications they would be suitable for. In fact, I didn't know whether they would have any applications in cryptography or cybersecurity.

Between 2007 and 2008, I decided to start applying for my first patent, the system that I later called MB09. At the time, I was studying some problems related to digital payments, and the goal was to implement an algorithm that could fit this kind of environment. When I finally completed the application for MB09, it was June 12, 2009. It is likely that at that time, only a few people knew about a new wave in *crypto-finance* guided by a group of cyber-punks and crypto-anarchists. This new wave was rising as one of the most innovative technologies invented in the 21st century.

With their headquarters in an artistic, funky building inside a pseudo-hotel located at 20 Mission Street in the heart of San Francisco, the newly born group set up a *hacker house* to meet and discuss the new technofinance. When I visited this place some years ago, the group had already left it, but the stories of epic brainstorming and meetups created a sense of mystery around this location.

Figure 6.1: The hacker house at 20 Mission Street, San Francisco: headquarters of Bitcoin (photograph by the author)

Among this group, there was a legendary figure called Satoshi Nakamoto, the mysterious author of a published white paper, titled *A Peer-to-Peer Electronic Cash System*, on digital payments. It was November 2008. That was also the beginning of a new era that was destined to forever change the digital payment system.

In the abstract of his white paper, based on previous ideas from WeiDai b-money, Satoshi stated the following:

> *"We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power."*

The paper never mentioned the name *blockchain*, but it is evident that it refers to it when it says *the longest chain*.

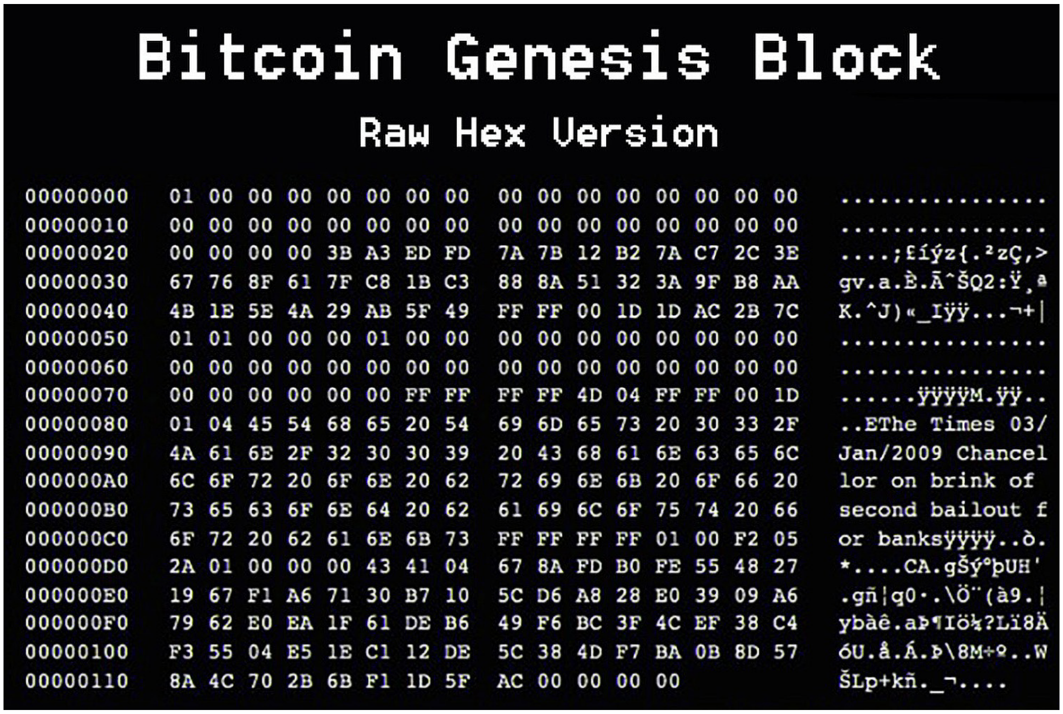
It was the prelude of the age of cryptocurrency, which culminated in the rise and common usage of one of the most extraordinary inventions in virtual money: Bitcoin. Nowadays, who hasn't heard about Bitcoin? Perhaps we know a little about Zcash (we discussed this in *Chapter 5*, *Zero-Knowledge Protocols*) and all the myriads of virtual coins released in the period of **Initial Coin Offering** (**ICO**). It was a correlated phenomenon linked to Bitcoin, something similar to an **Initial Public Offering** (**IPO**) but made up of cryptocurrency virtual tokens, starting in 2016.

This chapter will focus on algorithms related to secure and anonymous payments in the period between 2008 and 2012. It's essential to understand the context in which the idea of a new, free, and anarchic way of conceiving money has arisen. Additionally, it's important to understand the technology behind it: the blockchain. It was probably the fulcrum and the core of digital payment systems. Now, it's rising as one of the emergent technologies that enable many applications in FinTech, artificial intelligence, healthcare, and other sectors.

However, we have to go back a few years in this story. In *Chapter 4, Hash Functions and Digital Signatures*, I told you about David Chaum and his DigiCash, which was crafted in the 1990s. Blind signatures anonymized this form of digital payment. It was undoubtedly the first attempt to create digital money. In 2008, and precisely on the Wednesday after September 15 of that year, when the financial colossus Lehman Brothers collapsed, many other banks were also involved in bankruptcy and bailouts.

An extraordinary financial crisis called the *subprime mortgage crisis* laid the foundations for this new wave of cryptocurrency. The collapse of the financial system and people's deep uncertainty pushed Satoshi Nakamoto to re-think the way to produce, transmit, and spend money. Indeed, in his already mentioned white paper, the three pillars predicted by Satoshi Nakamoto to build the perfect currency are *no government*, *no banks*, and *no trusted third parties*. On January 3, 2009, when Satoshi crafted the first *genesis block* of Bitcoin, he sculpted in the *blockchain* a note next to the numbers generated by the first hash of the chain. This note read *Chancellor on brink of second bailout for banks* and remarked on his hatred of banks and the traditional financial system.

In the following screenshot, you can see the genesis block of Bitcoin as it appeared:



Bitcoin Genesis Block

Raw Hex Version

```
00000000   01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000020   00 00 00 00 3B A3 ED FD  7A 7B 12 B2 7A C7 2C 3E   ....;£íýz{.²zÇ,>
00000030   67 76 8F 61 7F C8 1B C3  88 8A 51 32 3A 9F B8 AA   gv.a.È.Ã^ŠQ2:Ÿ¸ª
00000040   4B 1E 5E 4A 29 AB 5F 49  FF FF 00 1D 1D AC 2B 7C   K.^J)«_Iÿÿ...¬+|
00000050   01 01 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
00000060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000070   00 00 00 00 00 00 FF FF  FF FF 4D 04 FF FF 00 1D   ......ÿÿÿÿM.ÿÿ..
00000080   01 04 45 54 68 65 20 54  69 6D 65 73 20 30 33 2F   ..EThe Times 03/
00000090   4A 61 6E 2F 32 30 30 39  20 43 68 61 6E 63 65 6C   Jan/2009 Chancel
000000A0   6C 6F 72 20 6F 6E 20 62  72 69 6E 6B 20 6F 66 20   lor on brink of 
000000B0   73 65 63 6F 6E 64 20 62  61 69 6C 6F 75 74 20 66   second bailout f
000000C0   6F 72 20 62 61 6E 6B 73  FF FF FF FF 01 00 F2 05   or banksÿÿÿÿ..ò.
000000D0   2A 01 00 00 00 43 41 04  67 8A FD B0 FE 55 48 27   *....CA.gŠý°þUH'
000000E0   19 67 F1 A6 71 30 B7 10  5C D6 A8 28 E0 39 09 A6   .gñ¦q0·.\Ö¨(à9.¦
000000F0   79 62 E0 EA 1F 61 DE B6  49 F6 BC 3F 4C EF 38 C4   ybàê.aÞ¶Iö¼?Lï8Ä
00000100   F3 55 04 E5 1E C1 12 DE  5C 38 4D F7 BA 0B 8D 57   óU.å.Á.Þ\8M÷º..W
00000110   8A 4C 70 2B 6B F1 1D 5F  AC 00 00 00 00            ŠLp+kñ._¬....
```

Figure 6.2: The genesis block by Satoshi Nakamoto

We will not analyze here all that takes place in processing a transaction in Bitcoin blockchain. We will specifically analyze the cryptography algorithms that lie at the foundation of it. The real difficulty for Satoshi Nakamoto was finding a way to perform the validation process for the transmission of digital currency, which is called the **consensus problem**. I will return to this concept of *consensus* later in this chapter to demonstrate that it is possible to avoid it, under certain conditions, using only the power of cryptography algorithms. So, I will suggest that blockchain is superfluous to process a digital transaction. Essentially, everything can be done using cryptography and avoiding blockchain.

Two other motivations pushed me to project algorithms in the field of public/private key encryption. The first was related to another payment method, called **M-PESA**, which I came across during my research (2007–2008). The second was related to my big wish to try to fight against the consolidated standard algorithms.

Let's look at the first motivation. I suppose that not many of you know what M-PESA is and how it works. It is a straightforward payment method, so I will not spend more than a few words explaining its mechanism. The interesting question is why do two-thirds of people born in Africa (more than 1.2 billion) use M-PESA to send money via cellphones and use it to receive the majority of the $500 billion sent from emigrants in other countries?

The answer is equally simple: low transaction fees (compared to MoneyGram and Western Union's higher fees) and the simple method of using the application on a phone. It's well known that in Africa there are more cellphones than cars, houses, or bank accounts. Africa will be the next colossal digital market for finance; it would be impossible to have enough bank branches or ATMs to cover the large area of the continent. So, the most widely used method of payment and transmission of funds is using a phone. This phenomenon attracted so much of my attention that I spent months investigating the systems of payment adapted for cellphones.

At the time, Vodafone (the big telecommunication company) owned 40% of Safaricom (Kenya's biggest telecom company). In 2007, Safaricom began a pilot program that allowed users to send money via cellphone. Vodafone had rolled out the product in Tanzania, South Africa, Mozambique, Egypt, Fiji, India, and Romania, beginning with Kenya itself.

In May 2011, after having implemented my first crypto-digital payment system, showing that it worked, I went to sign a contract with a big telecommunication company involved in digital payments (I have mentioned one of these previously). The agreement's purpose was to explore the possibility of implementing a property platform for digital payments. The telecommunication company also engaged my team and me in a new research project related to implementing another algorithm that was to become a new system: MBXI. Now, after many years, thinking back to that time, I can say that the payment system that I developed, implemented, and tested together with my collaborators (the engineers), Tiziana Landi and Alessandro Passerini, was one of the best things I have done in my career.

Even though the MB09 payment system has never been adopted as a standard, in the next section you will find some interesting properties in the algorithm that might be useful to increase your ability as a cryptographer.

Now, if you follow me, I will guide you in discovering the first of the two algorithms, MB09. This is a method based on Fermat's Last Theorem and remodeled in cryptographic mode. Then, we will analyze MBXI along with an interesting attack on RSA.

Introducing the MB09 algorithm and an attempt at demonstrating Fermat's Last Theorem

First, we will start with some considerations regarding the algorithm and reintroduce Fermat's Last Theorem. The first time I presented MB09, it was as an encryption algorithm, but effectively it is much more of *a protocol for digital payments*. As I have already mentioned, while blockchain and cryptocurrency were not yet well known, I developed MB09 as an encryption/decryption algorithm to exchange a message between two actors. Many years later, I worked on the algorithm, taking it as the basis for a *fully homomorphic encryption* system and creating MB23, which was a *fully homomorphic algorithm*. Eventually, in 2020, it was turned into a new version, called **MBXX**, to overcome the consensus problem proposed by Satoshi Nakamoto and its related technology: the Bitcoin blockchain.

Let's examine how the first version of the MB09 algorithm worked:

1. To do that, we'll recall Fermat's Last Theorem:

   $$a^n + b^n = z^n$$

   Here, the *n* exponents represent all the sets of positive integers.

   This equation, as we already know, has been demonstrated to be never satisfied except for the *n = 2* exponent (which is the well-known Pythagorean theorem), as follows:

   $$3^2 + 4^2 = 5^2$$

2. Diving deeper to analyze this simple yet complex equation, we find that in modular mode, by substituting the *n* exponents with a set of prime numbers, *p*, in the first equation, the following equation's results are always verified for the modular math addition property:

   $$a^p + b^p \equiv z^p \ (mod\ p)\ (for\ all\ p > 2)$$

   In simple words, the second equation shows that the sum of $a^p + b^p$ is always verified to be z^p for all primes *>2*. The proof of its verification is the following:

   - Fermat's Little Theorem states that $a^p \equiv a\ (mod\ p)$, $b^p \equiv b\ (mod\ p)$, and so on to *z*, where *p* is a prime number. Remember *p* is a prime number and it's the same parameter both in the elevation and in the module.

- From the sum elementary math property, we have:

$$a + b = z$$

- If we substitute *a* with a^p *(mod p)*, *b* with b^p *(mod p)*, and *z* with z^p *(mod p)*, it stands that $a^p+b^p \equiv z^p$ *(mod p)* is always verified.

3. Continuing with the result from the start of *step 2*, we can also conclude that, for all of the *(p)* exponents (at the opposite of its correspondent linear equation mode), the sum of $(a+b)^p$ is always verified, and it's equal to z^p. In other terms:

   $$a^p + b^p \equiv z \ (mod\ p)$$

   For example, consider the following modular sum:

   $$3^{17} + 5^{17} \equiv 8 \ (mod\ 17)$$

   So, we can re-write the preceding equation in *step 2* in the following form:

   $$a^p + b^p \equiv (a + b)^p \ (mod\ p)$$

   Alternatively, for the modular sum property, we can even write it in the following simplified form:

   $$a + b \equiv z \ (mod\ p)$$

   That means if we put a prime number *(p)* as an exponent and even the modulus with the same *(p)* value, then the preceding equation (in modular mode) returns to linear mode, so that *(a+b)* is always equal to its sum, *(z)*.

At this point, I could hazard a demonstration:

1. In *step 3*, we reached the point of rewriting the equation to $a^p + b^p \equiv z \ (mod\ p)$.
2. In *step 2*, we demonstrated (following Fermat's Little Theorem) that the equation results are always verified for all *p>2*.

   For example, if we try *p =11*, we have:

   $$3^{11} + 5^{11} \equiv 8 \ (mod\ 11)$$

3. The sum of $a^p + b^p$ is always $z = a + b$, as we have previously seen. So, this equation (in modular mode) is always verified to be valid in the opposite way of what we want to demonstrate in linear mode: the sum $a^n + b^n$ must always be denied z^n.

4. But still, we have a problem: the exponent n is an integer number and not just a prime number p. Is this a real problem? If I demonstrate that it is valid for any p, is it valid for all the integer numbers?
5. Now let's assume by contradiction that the following equation (putting *p'* instead of *p* as the module) would always be verified:

$$a^p + b^p \equiv z^p \ (mod\ p')$$

Where *(p')* is the set of all the prime numbers > z^p. So, by definition, we can also take *p'>z*.

Can you catch the meaning?

If $p' > z^p$, we are dealing with a sum of $a^p + b^p$ expressed in linear mode, no longer in modular mode, because the operation performed with the sum falls back inside the ring *(Zp')*.

For example, let's assume that the first prime number after $z^p = 8^{11}$ is *p'*:

$$p' = 8589934609$$

So, we calculate the equation in *(mod 8589934609)*:

$$3^{11} + 5^{11} = 8^{11} = 8589934592\ (mod\ 8589934609)$$

There is no difference in performing such an equation in modulo *p'* or in linear mode.

6. In fact, both equations will never be satisfied.

That's because for Fermat's Little Theorem, again, we have:

$$z^p = z\ (mod\ p)$$

Consequently:

$$z^p \neq z\ (mod\ p')$$

As a result, in *step 5* of this demonstration, we had wrongly assumed $z = a^p + b^p\ (mod\ p')$.

So, now we must assume that:

$$z^p \neq a^p + b^p\ (mod\ p')$$

This last equation is equivalent to writing:

$$z^p \neq a^p + b^p$$

That was what we wanted to demonstrate for the Fermat version with prime exponents.

Now, returning back to MB09, the following equation is what Fermat's Little Theorem explains, and it determines the preceding linear function:

$$a^p = a \ (mod \ p)$$

For example, consider the following:

$$3^7 = 3 \ (mod \ 7)$$

You might not have noticed, but in the preceding function, we used the equal symbol (=), not the symbol ≡, which means congruent notation. In this case, the meaning of = is very important because we mean equal in its absolute values, so 3 is actually 3, and not just congruent.

I understand that all this could appear a little bit fuzzy, but try to follow me a little bit further ahead and the fog should disappear.

Fermat's Little Theorem states that $a^p = a \ (mod \ p)$ if $a<p$. So, what happens if we assume $a>p$?

For example, let's try with $a = 5$ and $p = 3$:

$$5^3 \equiv 2 \ (mod \ 3)$$

As you can see, (a) is no longer equal to (a) in its absolute value if a>; the result is only congruent.

This intriguing property caused me to bear many considerations in mind regarding Fermat's Little Theorem, the most important of which here is the formulation of the following hypothesis:

"If I take a>>p (much greater than p) and I keep [a] secret, then this will be a one-way function where it will be very difficult to return from the result, let's say from (A) to [a]."

Let's look at an example.

You can verify that $5^3 \equiv 2 \ (mod \ 3)$, but also $8^3 \equiv 2 \ (mod \ 3)$, and 11^3 ....14^3, 17^3 are all congruent, and so on infinitely.

As you might have gathered, the curious thing is that starting from an initial value of 5 and adding *3 + 3 + 3…* (essentially, adding 3 sequentially), we always obtain the same value (2) in *(mod 3)*, which is different in absolute value from (5), as Fermat's Little Theorem states when $p > a$.

Finally, at the time, I wanted to demonstrate that it is very difficult to come back from a public number (A) to a secret number *[a]* if $p < a$.

Let me explain this concept with an example taking a larger *[a]*:

$$[a] = 96269369030336679694019965478670$$

That is, it would be easy to reduce this number to its corresponding modulo *p=3*.

In fact, we have the following:

$$96269369030336679694019965478670 \equiv 2 \ (mod\ 3)$$

Conversely, if I know the public number, *(A) = 2*, where the following is true:

$$A \equiv [a]^p \ (mod\ p)$$

Then, even if an attacker knows *p = 3*, it will be very difficult to attempt to obtain the *[a]* private key if *a>>p*:

$$[a] = 96269369030336679694019965478670$$

The simple reason for this is that *p = 3* is our *(Z^p)* ring, but *[a]*, the private secret key, is outside of the *(Z^3)* ring, and you don't know when to stop the iteration to find number *[a]*. Theoretically, *[a]* could be all the numbers from *(p)* to infinity and it doesn't matter how large *[a]* is because, computationally, it is very easy to obtain (*A*) from *[a]* but not vice versa. However, all that works but only in some instances.

Now that you know the basis on which MB09 leans, you will probably be curious to learn how it was implemented in its first version.

Important note

I have represented, with a lowercase *[a]* letter, the hidden elements of the equation and, with the uppercase *(A)* letter, the known elements. However, don't be confused by this, and don't assume that *A* > *a* just because the letters are lower and uppercase. In fact, it's the opposite: in mathematical terms, *A* is much smaller than *[a]*.

An extensive explanation of the MB09 algorithm

As I mentioned earlier, the MB09 system is based on public/private key encryption principles. However, the scope here is *not* to send and receive a message between two (or more) actors. MB09 doesn't work correctly if used for this scope. However, the scheme of the algorithm works well for the scope to set up a protocol to manage and transmit digital cash in anonymity and secure the transactions.

Eventually, one of the purposes of implementing such a system is that the network works as an *autonomous system*. I will explain this concept better in the next section when introducing MBXX, which is the evolution of MB09 in a decentralized environment. The version I'll present now, starting from a centralized system, will migrate to a decentralized and distributed system, as we will discover in MBXX.

In the next section, I will explain the basic concepts of a network governed by crypto algorithms, where *Z* (the centralized administrator) must verify whether all the transactions made by the users are correct and acceptable.

Z is our network's *centralized administrator*; you can think of it as a telecommunication provider or a bank. Alice and Bob are two actors in the network.

Bob is the sender, and Alice is the receiver. They can be considered virtual machines, servers, or computers. In our basic example, they are part of a network in which there are many users. The network's admin (*Z*) is a server that is linked to many users, as you can see in the following diagram of a centralized network:

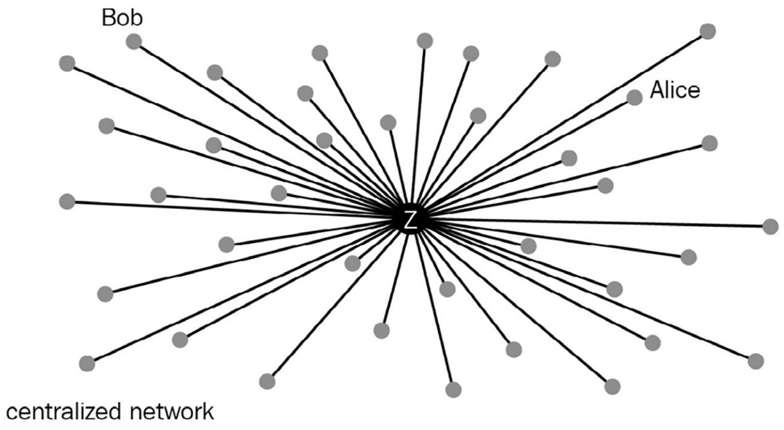


Figure 6.3: A centralized network

Z has already published some parameters such as the prime number *(p)* and the public keys, *(A)* and *(B)*, of Alice, Bob, and other participants.

We will proceed to look at the algorithm with only these two actors. Still, as I have already mentioned, this algorithm's strength is just the possibility of operating with many users who don't know each other, who don't trust each other, and who don't trust third parties.

Indeed, Fermat's Last Theorem (which is extended in modular form), in this case, applied to MB09, is valid also for an infinite number of users of the network, as you can mathematically see here:

$$a^p + b^p + c^p + n^p \equiv z^p \ (mod\ p)$$

Important note

I have illustrated the operations with a sum *(+)*; however, all of these operations could be performed by *XOR*, difference, or multiplication.

The *[M]* message has previously been encoded with ASCII code (as we learned in the first chapter), and you can think of *[M]* as an amount of digital money to transfer between *(A)* and *(B)*.

Now, let's demonstrate whether Alice and Bob can transfer money to each other using the algorithm mathematically.

Each step of this algorithm after the *key initialization* consists of a recursive iteration of the following:

- Implementation of the public parameters of the users
- Operations on the parameters: the transmission of the messages (for example, a digital money transfer)
- Re-valuation of the new parameters

Key initialization:

- [a]: This is the private secret key of Alice.
- [b]: This is the private secret key of Bob.

This assumes that *[a]* and *[b]* are two random large primes.

Public parameters:

Alice and Bob calculate their public keys starting with their private keys:

- A ≡ a (mod p) [a >> p]
- B ≡ b (mod p) [b >> p]
- C ≡ c (mod p) [c >> p]
- Z ≡ z (mod p) [z >> p]

At this point, you can visualize the two *f(z)* and *f(Z)* equations. The first is in modular mode, and the second is transformed into linear mode:

$$a^p + b^p + c^p + \cdots + n^p \equiv z^p \pmod{p}$$

$$A + B + C + \ldots + N = Z$$

The correspondence between the first equation's elements and the second equation's elements is essential for the algorithm. As you can observe in the following diagram, each element of the first equation can be represented with an element of the second equation.

The peculiarity of this correspondence is that, while in the first equation, the operations have been performed using the users' private keys; in the second equation, the operations are performed with the public keys of the users. You can see that the arrows are only going in one direction, from *[a]* to *(A)*, and it's difficult, as we have seen, to go back from *(A)* to *[a]* if *[a>>p]*:

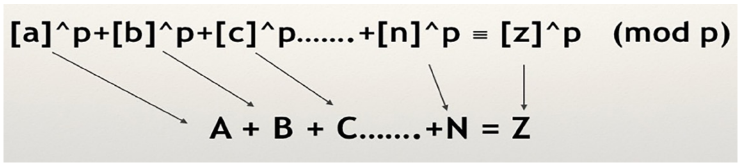


Figure 6.4: The correspondence between the elements of two equations

Here, *(Z)* can be defined as the *isomorphic balance* between the operations performed with the parameters expressed clearly, instead of inside the square brackets, representing the secret parameters of the *f[z]* function.

At this point, MB09 (version 2009) went on with a standard cryptographic transmission based on a pair of keys to transmit a message between Alice and Bob.

Important note

Remember that, here, *[a]* and *[b]* are Alice and Bob's secret values. Moreover, don't confuse the letters *[a]* and *(A)*, thinking that *(A)* is bigger than *[a]* just because the first is represented in capital letters.

First, we have to generate the secret transmission key, *[K]*. This is a shared transmission key that can transmit *[M]* between Alice and Bob. It can be generated with any public/private key algorithm, just like the D-H algorithm.

Indeed, in the first version of MB09, I adopted the *[K]* key generated by D-H. However, as you might have gathered, it's possible to generate *[K]* with any other algorithm that uses the *[a]* and *[b]* secret keys of Alice and Bob as input and gives a cryptogram, *(c)*, as output.

Important note

(c) is the cryptogram represented inside round brackets. Here, I use *(c)* because we have only two actors. In other cases, I will use another notation; don't confuse it with an element of the network.

The representation of the algorithm with only two actors (but as I said, this is a multiple-communication algorithm) is as follows:

$$a^p + b^p \equiv z^p \ (mod\ p)$$

$$(A) + (B) = (Z)$$

Alice encrypts *[M]* using her *[K_a]* secret key, where *[K_a]* has been generated by any public/private key algorithm, and obtains the cryptogram *(c)*:

$$[M] * [K_a] \equiv (c)\ (mod\ p)$$

Important note

Here, as I have already mentioned, *(c)* is the cryptogram, which is obtained through the multiplication of the secret message *[M]* with Alice's private key *[K_a]*. In the original version of the algorithm, I used multiplication as the operation to generate *(c)*. However, it is possible to implement it with a different encryption method, such as Bitwise-XOR or scalar multiplication.

Bob decrypts *[M]*, generating a *[K_b]* secret key such as the following:

$$c\ (INV)\ [K_b\ ] = [M]\ (mod\ p)$$

For instance, if we assume the use of this algorithm in a digital cash environment, such as the transmission of digital cash in a payment system, *[M]* will be the real amount of digital cash transacted, while *(Hm)*, its corresponding hash value, will represent the hash of *[M]*:

$$[a\ +/-M]^p + [b\ +/-M]^p \equiv [z\ +/-M]^p\ (mod\ p)$$

$$(A \ +/- \ Hm) \ + \ (B \ +/- \ Hm) \ = \ (Z)$$

The operations performed on the first equation (at the top), encrypted, give the result *[z]* linked by an operation to the message *[M]* "in blind" that corresponds with the operations performed by the second equation's elements and matches the result of the linear equation *(Z)*. We will experiment with a similar protocol later in this chapter when we discuss the MBXX protocol.

Essentially, *(Z)* represents the *homomorphic balance* of the system, as the digital money transacted doesn't change the value of *(Z)*. The scope to adopt such a homomorphic balance is that the administrator has, in each instance, a corresponding balance of the total amount of cash with the *isomorphic values* and can control the accuracy of the transactions even when the transacted amount, *[M]*, is unknown. If you're struggling to understand *(Z)*, note that we will be working with (Z) again when we look at the MBXX protocol, particularly in the *Notes on the MBXX protocol* section.

Let's perform an exercise using a D-H key exchange:

- *[a]* and *[b]* are the secret parameters of Alice and Bob.

As we know, the public parameters in D-H are as follows:

- *p*: Large prime number
- *g*: The generator of the (Z^p) ring

The exchanging of *[M]* is performed as follows.

Step 1: *Encryption*

Alice generates her public key *(Ag)* starting with her secret parameter, *[a]*:

$$Ag \ \equiv \ g^{[a]}(mod \ p)$$

Bob generates his public key *(Bg)* starting with his secret parameter, *[b]*:

Alice calculates the following:

$$Bg^{[a]} \ \equiv \ [K_a] \ (mod \ p)$$

Alice encrypts *[M]* with her $[K_a]$ key:

$$[M] \ * \ [K_a] \ \equiv \ (c) \ (mod \ p)$$

Step 2: *Decryption*

Alice sends (c, Ag) to Bob.

Alice sends the hash value of *[M] = (Hm)* to the administrator.

Bob decrypts (c), returning *[M]*:

$$(c) * [(INV)K_b] \equiv [M] \; (mod\; p)$$

Note that $[(INV)K_b]$ is just the inverse modular operation of multiplication. You can figure it out like a division: $c/K_b = M \; (mod\; p)$. You can refer to *Figure 6.5* for a complete scheme of the algorithm applied to the transmission of a secret message.

If we consider the private keys of Alice and Bob, *[a]* and *[b]*, as values of their accounts and *[M]* as the amount of digital cash transmitted, we have a third step, which can be regarded as the homomorphic balance.

Step 3: The homomorphic balance

Transposing the value of *[M]* both into the first equation and the second equation, the admin can verify that the transaction between *(A)* and *(B)* is correct:

$$[a \; +/- M]^p + [b \; +/- M]^p \equiv [z \; +/- M]^p \; (mod\; p)$$

$$(A \; +/- Hm) + (B \; +/- Hm) = (Z)$$

Pay attention to the following in the function of the encryption:

$$[M] * [K] \equiv (c) \; (mod\; p)$$

As I have said, I have used the multiplication between *[M]* and *[K]* to encrypt *[M]*. However, we can also implement the encryption with another operator, such as Bitwise XOR, between the message *[M]* and the key *[K]*. Note that, here, the algorithm of the transmission adopted to send *[M]* is not as important as the logical structure of the protocol for the implementation of multiple transmissions.

Indeed, the logical structure of this algorithm is not crafted for two users but multiple users.

In this scheme, it is supposed that the admin knows the amount of money originally held by the users. Let's look at an example:

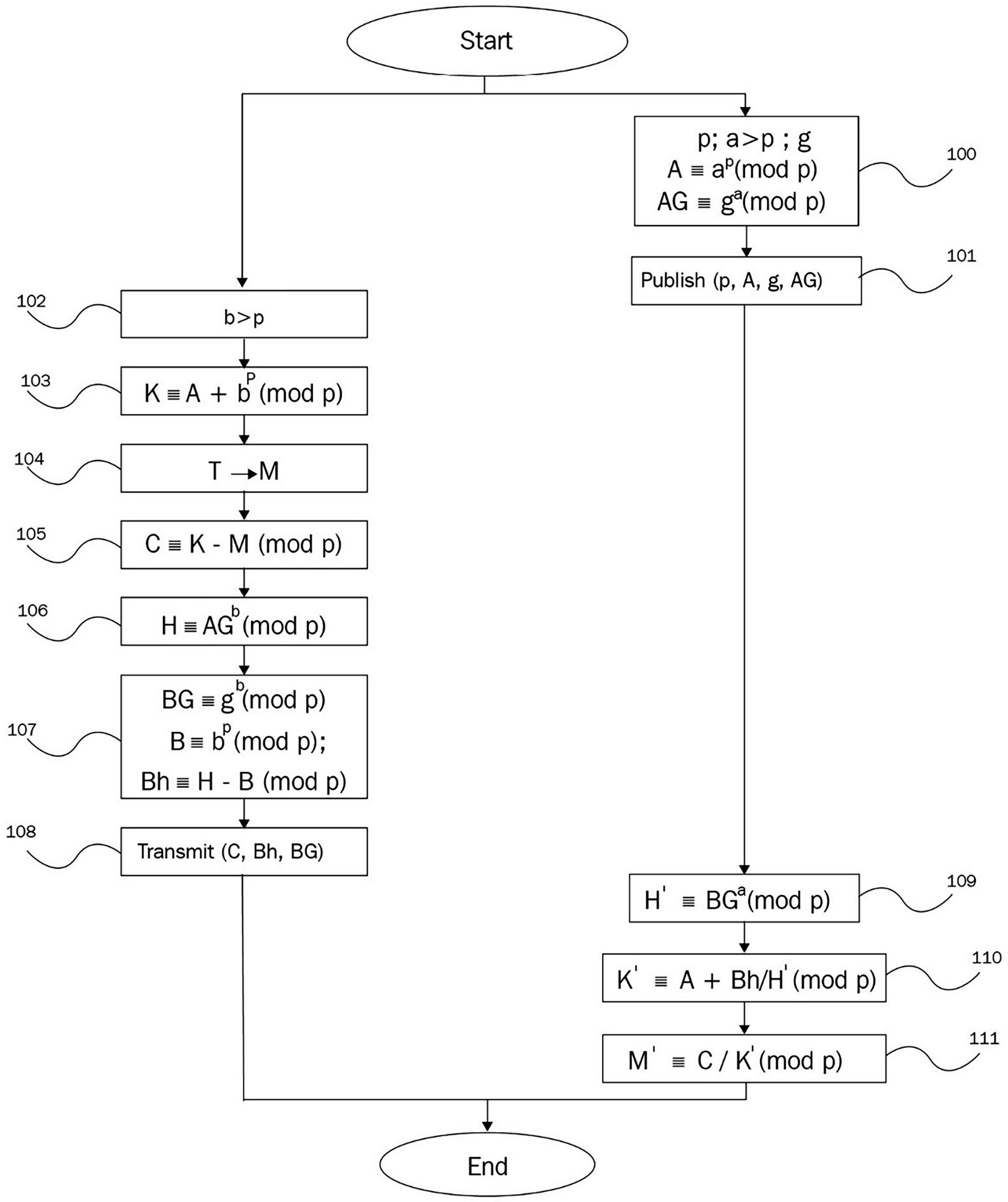

Start
p; a>p ; g
$A \equiv a^p (\mathrm{mod}\ p)$
$AG \equiv g^a (\mathrm{mod}\ p)$
100
Publish (p, A, g, AG)
101
102
b>p
103
$K \equiv A + b^P (\mathrm{mod}\ p)$
104
$T \rightarrow M$
105
$C \equiv K - M\ (\mathrm{mod}\ p)$
106
$H \equiv AG^b (\mathrm{mod}\ p)$
107
$BG \equiv g^b (\mathrm{mod}\ p)$
$B \equiv b^p (\mathrm{mod}\ p);$
$Bh \equiv H - B\ (\mathrm{mod}\ p)$
108
Transmit (C, Bh, BG)
109
$H' \equiv BG^a (\mathrm{mod}\ p)$
110
$K' \equiv A + Bh/H' (\mathrm{mod}\ p)$
111
$M' \equiv C / K' (\mathrm{mod}\ p)$
End

Figure 6.5: The original version of the patented MB09 published in the Patent Cooperation Treaty (PCT)

At the time, together with my colleagues, we crafted a **Proof of Concept** (**POC**) to demonstrate that MB09 could work well to exchange digital cash. We implemented a transmission of digital crypto cash between cellphones. The protocol used was based on a modified version of MB09, which I have presented here. It was added to the operative elements to engineer implementation.

In May 2010, the payment system based on MB09 was selected as a candidate for digital payments by a telecommunication company. This company then acquired part of the rights on the patent of MB09 and entrusted our research laboratory with a research project to implement another private/public key algorithm, MBXI, which we will examine next.

Introducing the MBXI algorithm

When I started the MBXI project, it was 2010. I had already had the algorithm's scheme in my mind for some time before that, but there were some issues that I couldn't solve. While I was working on project MB09, to build a digital payment network, the telecommunication company that became my partner also needed to use a proprietary algorithm to send a message, *[M]*. In this scenario, that refers to the amount of digital money transmitted between two or more actors in the network.

So, I proposed a new public/private key algorithm. MBXI was patented in November 2011, and it's still valid now after ten years.

MBXI is a cryptographic process method that could be simultaneously considered both an asymmetric scheme and a symmetric scheme. We learned in *Chapter 2*, *Symmetric Encryption Algorithms*, how symmetric schemes work, and in *Chapter 3*, *Asymmetric Encryption Algorithms*, we learned how asymmetric schemes work. This algorithm bases its strength on the discrete logarithm problem, which has already been explored across different algorithms in this book (for example, D-H, ElGamal, zero-knowledge protocols, and more). As mentioned, the discrete logarithm problem is still a very hard problem to solve; we have learned that if modular equations (such as the *p* modulo) are implemented using a big prime number, the solution of exponential modular equations is very burdensome or almost impossible.

This algorithm derives its robustness from the application of modular exponential equations, which are injected (one-way) functions for defining exponents of the encryption (and decryption) equations. As you can see in the flowchart in *Figure 6.6*, the *[T]* message is decomposed into *[M]* sub-messages divided into blocks.

In MBXI, the only way to determine the private key is to solve the discrete logarithm differently with other asymmetric encryption algorithms, such as RSA, that suffer from both the problems of factorization and discrete logarithm (as explained in *Chapter 3*, *Asymmetric Encryption Algorithms*), along with ElGamal.

In fact, the best way to define this algorithm is as neither asymmetric nor symmetric but a public/private key algorithm; I will explain the reasons for this later in this chapter.

Let's dive deeper to explore the scheme of MBXI.

Alice and Bob are always our two actors. In this scenario, let's consider that Bob wants to send an *[M]* secret message.

The first step of the algorithm is to generate the private and public keys given by the common public parameters published on the network:

- *p*: This is a large prime number.
- *g*: This is a generator of the ring (*Zp*).

Step 1: *Key generation*

- *[a]*: This is the private secret key of Alice (a large number chosen inside p-1).
- *[b]*: This is the private secret key of Bob (a large number chosen inside p-1).
- Alice's public key: $K_A \equiv g^a \pmod{p}$.
- Bob's public key: $K_B \equiv g^b \pmod{p}$.

Step 2: *Encryption*

The encryption in MBXI is done by an inverse modular equation expressed by solving the following function:

$$\{[{K_a}^b + E_B]\ (mod\ p)\} * x \equiv 1\ (mod\ p - 1)$$

In this inverse equation, Bob takes the public key of Alice, *(K_A)*, elevates it to his private key, *[b]*, and adds it to *(e_B)*, where *(e_B)* is a parameter selected from the group of integers so that the co-primeness between the following is verified:

$[{K_a}^b + E_B]\ (mod\ p)$ and $(p-1)$

Then, after solving the inverse modular equation for *[x]*, Bob calculates the cryptogram, *(C)*, which is given by the function:

$$C \equiv M^x\ (mod\ p)$$

Bob sends the triple *(C, e_B, K_B)* function transmission to Alice.

Step 3: *Decryption*

Alice can now decrypt (C) using Bob's public key, (K_B), and her private key *[a]*, along with the parameter *(e_B)*. This parameter is transmitted by Bob, whose function will be better described later during the analysis of the algorithm.

The decryption consists of solving the following function, which produces *[y]* as Alice's private decryption key:

$$y \equiv \{[{K_a}^b + E_B]\ (mod\ p)\ (mod\ p)\}$$

Finally, the *[M]* message is returned to Alice, elevating *(C)* to *[y]*:

$$M \equiv c^y\ (mod\ p)$$

Where, as mentioned, *[y]* is Alice's private decryption key, represented by the solution of the preceding inverse modular equation.

The flowchart in the following diagram will help you gain a better understanding of the entire process of the encryption/decryption of MBXI:

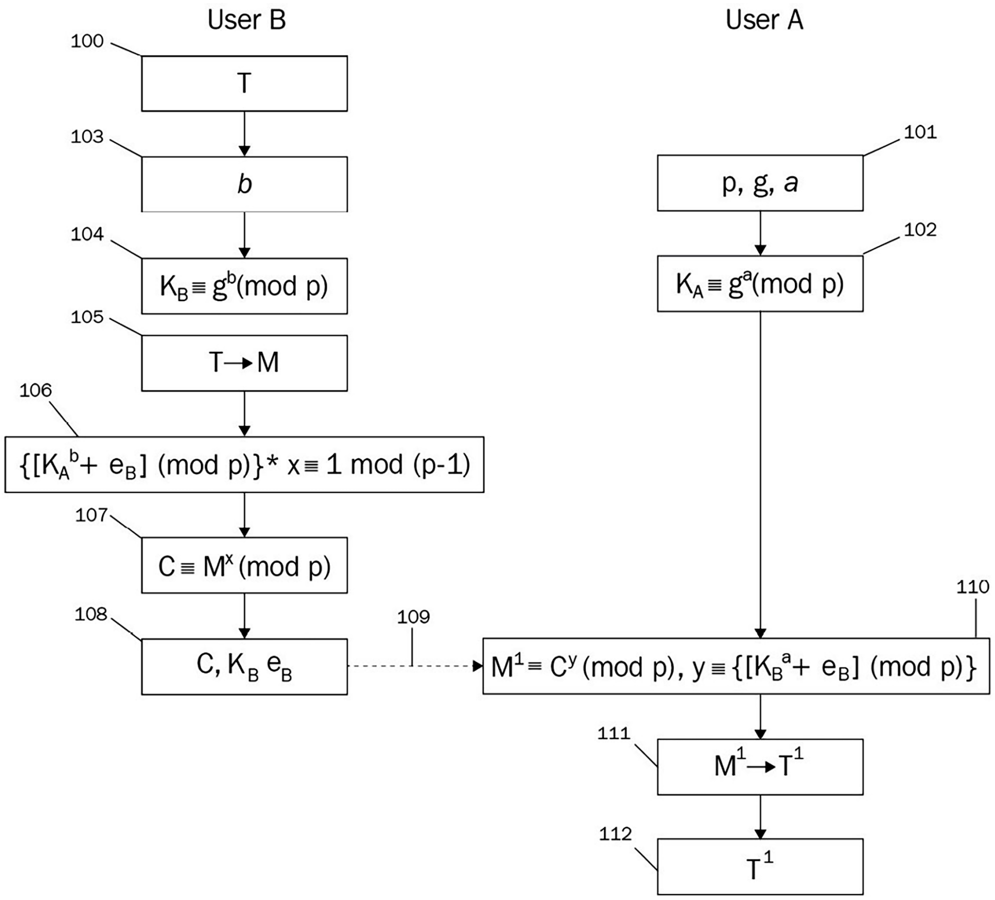

User B
User A
100
T
103
b
104
$K_B \equiv g^b (\text{mod } p)$
105
T→M
106
$\{[K_A{}^b + e_B] (\text{mod } p)\} * x \equiv 1 \text{ mod } (p-1)$
107
$C \equiv M^x (\text{mod } p)$
108
$C, K_B\ e_B$
109
101
p, g, a
102
$K_A \equiv g^a (\text{mod } p)$
110
$M^1 \equiv C^y (\text{mod } p), y \equiv \{[K_B{}^a + e_B] (\text{mod } p)\}$
111
$M^1 \rightarrow T^1$
112
T^1

Figure 6.6: The encryption/decryption process in MBXI

Now that we have examined how MBXI works, let's look at an example with numbers to understand it better.

A numerical example of MBXI

The following numerical example represents a cryptographic communication established between a sender (Bob) and a receiver (Alice). Of course, like all the other examples, we use relatively small numbers. Bear in mind that, in a real application, the private keys are, at the very least, of the order of 3,000 bits (that is, thousands of digits), such as the prime (*p*).

We will begin with the following two assumptions:

- *p = 7919*.
- *g = 7* is the generator.

Alice selects her private key, *[a]*:

$$a = 123456$$

Bob selects his private key, *[b]*:

$$b = 543210$$

Important note

As you have noticed, I have selected a sequence of numbers for *a = 123456* and *b = 543210*. This is just an example, and you should never use a sequence of numbers when you select a password in the real world because this is the first combination of numbers tried by an attacker.

According to the first step of the algorithm (key generation), Alice computes her public key as follows:

$$K_A \equiv 7^{123456}(mod\ 7919) = 7036$$

Bob does the same, and computes his public key as follows:

$$K_B \equiv 7^{543210}(mod\ 7919) = 4997$$

Bob's encryption is performed in accordance with the following equation:

$$\{[7036^{543210} + 1\,]\,(mod\ 7919)\} * x \equiv 1\,(mod\ 7919 - 1)$$

Here, $(e_B)= 1$ is the smallest integer that verifies the equation.

Therefore, Bob determines the following private encryption key *[x]*:

$$x = 3009$$

With this parameter, Bob calculates the cryptogram, (*C*):

$$C \equiv 88^{3009}\,(mod\ 7919) = 2760$$

Bob sends the triple *$(C, K_B, e_B) = (2760, 4997, 1)$* transmission to Alice.

Once Alice has received the triple *(C, K_B, e_B)* transmission, she is able to decrypt cryptogram (*C*) and determine the message block of *[M]*:

$$M = 2760\hat{}\{[4997^{123456} + 1]\ (mod\ 7919)\}\ (mod\ 7919) = 88$$

Indeed, *M = 88* matches the original message block, *[M]*, transmitted by Bob.

As you can observe, in the preceding example, I have demonstrated with real numbers that the algorithm works. Now, I will offer some notes on it to point out some functions and show similarities with RSA.

Notes on the MBXI algorithm and the prelude to an attack on RSA

First of all, I want to clarify the meaning of the *(e_B)* parameter. This is clearly transmitted between the sender and the receiver. However, in different versions, it can be randomly processed.

For example, *(e_B)* is randomly selected from the range of (1, 10). Let's assume that *(e_B)* is *6*. Therefore, the equation of encryption will be as follows:

$$\{[7036^{543210} + E_B]\ (mod\ 7919)\} * x \equiv 1\ (mod\ 7919 - 1)$$

Note that for *$(e_B) = 6$*, the equation is *not* verified!

Indeed, if we try to put *$e_B = 6$* in the preceding function instead of *$e_B = 1$*, using the *Reduce* function in Wolfram Mathematica, this is the result:

$$Reduce[4960 * x == 1, x, Modulus -> 7919 - 1]$$

$$False$$

In other words, the inverse number for *4960 (mod 7919)* doesn't exist. In addition to this, the number that is randomly selected from the given range is increased by 1, and the check process is repeated using *$(e_B) = 7$*, which satisfies the co-primeness condition whereby *x = 5575*.

So, we will have the following encryption process:

$$C \equiv 88^{5575}(mod\ 7919) = 2195$$

In this case, Bob's triple transmission to Alice will be *$(C, K_B, e_B) = (2195, 4997, 7)$*.

So, the decryption will be as follows:

$$M \equiv 2195\text{^}\{[4497^{123456} + 7]\ (mod\ 7919)\}\ (mod\ 7919)$$

From this equation, *M = 88* matches the original message block of *[M]* that was transmitted by Bob.

Another important consideration relates more to the implementation of MBXI and, in particular, to several problems related to the length of the *[M]* message.

The *[M = 1]* message gives the ciphertext of *(C = 1)*, no matter what the encryption key.

Similar properties are true of RSA in its simplest form. Most RSA used in practice includes message padding, which eliminates these properties, and the same could be done with MBXI. For some very specialized uses, the preceding properties could be helpful; the technical term is *homomorphic encryption*. We will analyze the partial homomorphic property of RSA in *Chapter 8*, *Homomorphic Encryption and Crypto Search Engine*.

As discussed in *Chapter 5*, *Zero-Knowledge Protocols*, *padding* is necessary to remove the (generally) undesirable properties. This means that a ciphertext string will be longer than the underlying message string.

You might be wondering whether MBXI is an authentic asymmetric encryption algorithm. My answer to this question is no; MBXI is a public/private key encryption algorithm but not a pure asymmetric encryption algorithm like RSA. Even if this peculiarity of MBXI has some interesting applications, for example, to share a key, it is absolutely not in the scope of MBXI to substitute the power of a symmetric algorithm such as AES.

To demonstrate the scheme of the shared key, I will reformulate the equation that determines the encryption step:

$$C \equiv M^x (mod\ p)$$

Suddenly, we notice a similarity with RSA's encryption structure:

$$C \equiv M^e\ (mod\ N)$$

It's just a matter of swapping the *[x]* parameter with *(e)*, and it turns out that the two encryption schemes are similar.

In RSA, remember that *(e)* is a public parameter, and *(N)* is the public key of the receiver, given by *N = [p]*[q]*.

In contrast, in MBXI, *[x]* is a secret encryption key (more similar to D-H), and *(p)* is a big public prime number.

Anyway, these comparisons will be useful to explain a couple of interesting attacks on RSA that have inspired me due to the similarity of the RSA and MBXI encryption schemes.

Unconventional attacks and self-reverse decryption on RSA

The notion of an asymmetric backdoor was introduced by Adam Young and Moti Yung in their paper, *Proceedings of Advances in Cryptology*. An asymmetric backdoor can only be used by the attacker who plants it, even if the full implementation of the backdoor becomes public (for example, via publishing, being discovered and disclosed by reverse engineering, and more). This class of attacks has been termed kleptography; they can be carried out on software, hardware (for example, smartcards), or a combination of the two. The theory of asymmetric backdoors is now part of a larger field called **cryptovirology**. Notably, NSA inserted a paragraph on kleptographic backdoors into the Dual EC DRBG standard.

There exists an experimental asymmetric backdoor in RSA key generation. This OpenSSL RSA backdoor, designed by Young and Yung, utilizes a twisted pair of elliptic curves and has been made available. In this section, we will examine something that is a little bit different because it involves the injection of a parameter inside the *modulo N* of the RSA encryption function. Likewise, it's possible to modify a few rows in the code of the OpenSSL library, and as a result, we will achieve the creation of a backdoor in RSA.

A backdoor is something that allows an attacker to spy on the communications between two or more people, and in cryptography, a backdoor allows the attacker to decrypt an unbreakable cipher by adopting a mathematical or physical implementation method.

In both cases, creating a backdoor needs a certain allocation of resources in terms of skills and preparation to inject the malware. Eventually, the result is to allow a third party (Eve) to spy on or modify the encrypted message. However, there is also another possibility to create an even more malicious backdoor than others: the sender himself (Bob) might have an interest in injecting a backdoor in his own encryption.

Important note

Implementing a backdoor for malicious purposes is a crime. So, I am warning you to consider this section as merely a theoretical way to understand that there is the possibility of implementing a backdoor in RSA.

It's time to return to a question posed in *Chapter 3, Asymmetric Encryption Algorithms*: is this method of *self-reverse* decryption an unreasonable model, and not representative of practical situations, or does it, instead, have practical uses? In other words, why or in which cases could it be reasonable to use a method like this to attack a network?

You can think of Bob as the administrator of a telecommunications company, a social network, or any cloud provider who is willing to spy on communications between users. But even in the cryptocurrency space, there should be interesting cases if Bob, the sender (in the case of a cryptocurrency payment), is able to perform a fake signature on the amount transmitted. He will probably be able to perform *double-spending* of this digital money. Finally, the self-reverse decryption method could also be used (in this case, for a good purpose) to retrieve the encrypted files of a ransomware attack.

As Simon Singh wrote in his 1998 book *The Code Book*, a report on the *Wayne Madsen Report* blog revealed that a Swiss cryptographic company, Crypto AG, had built backdoors into some of its products and had provided the US government with details of how to exploit these backdoors. As a result, the American government was able to read the communications of several countries. In 1991, the assassins who killed the exiled former Iranian prime minister Shapour Bakhtiar were caught; this was thanks to the interception and backdoor decipherment of Iranian messages encrypted using Crypto AG equipment.

Now, let's explore how to create a logical backdoor using some of the functions we have learned about in this book.

We'll start with one of the main paradigms and constraints in the RSA encryption stage. This paradigm states that once the cryptogram has been sent by Bob, the sender, it cannot be re-decrypted by the sender. That happens because only the receiver (Alice), the owner of the *(p,q)* private keys, can perform *(N)*. Consequently, she is the only one who can decrypt the (*c*) cryptogram and receive the message *[M]*.

Let's refresh our knowledge from *Chapter 3*, *Asymmetric Encryption Algorithms*. In the following diagram, you can see the RSA algorithm encryption stage:

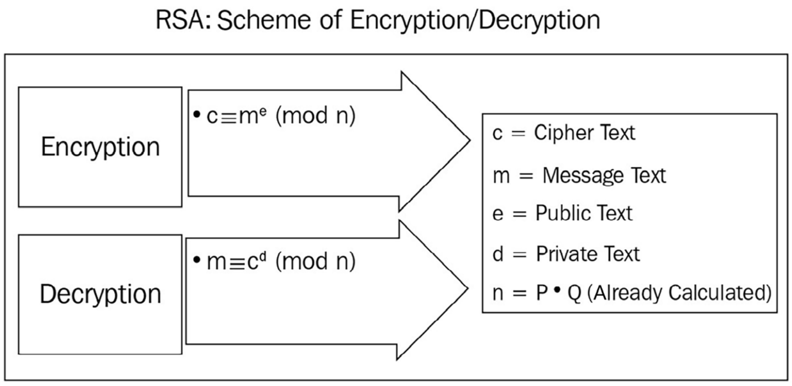


Figure 6.7: The RSA encryption stage

In this stage, we have the following:

- *[M]* is the secret message
- *(N)* = p*q is Alice's public key
- *e* is a public parameter (given)

I would remark that the encryption is performed by the sender (Bob) using Alice's public key, which we will now rename (*Na*) to clarify that it is Alice's public key:

$$[M]^e \equiv c\ (Mod\ Na)$$

Bob, the sender, knows the secret message, *[M]*, but technically, he is *not* able to *re-decrypt* mathematically the cryptogram (*c*) once he delivers the cryptogram (*c*). That's because it's assumed that only Alice knows her private key *[da]*.

Key generation

Alice's public key, *(Na)*, is given as follows:

$$Na = p * q$$

[da] is given as follows:

$$[da] * e \equiv 1\ (mod\ [p-1] * [q-1])$$

Recall from *Chapter 3*, *Asymmetric Encryption Algorithms*, that this last inverse multiplication is the core of RSA's algorithm because it allows you to retrieve the secret message, *[M]*, hidden inside the cryptogram, (*c*), through the following operation of decryption:

$$M \equiv c^{da} \ (mod\ Na)$$

Now it should be clear that one of the main paradigms of the RSA algorithm is that the cryptogram, (*c*), once it has been delivered by the sender, cannot be decrypted by anyone who isn't the holder of the *[da]* private key.

However, I can demonstrate that this statement is not applicable in some particular circumstances when a backdoor is injected.

Now, let's move on to analyze how to implement this pseudo-attack, which is able to generate self-reverse decryption (by Bob) of the cryptogram in RSA.

Remember again, here, we assume that Bob himself (the sender) is the author of the attack. Then, as you will see, it's easy to demonstrate that Eve, an external attacker, can also replace Bob.

The public and private keys

- *(Na)*: This is the public key of Alice (the receiver).
- *(Nb)*: This is the public key of Bob (the sender).
- *(e)*: This is a public parameter (given by the system).
- *[da]*: This is Alice's private key.
- *[db]*: This is Bob's private key.

Bob performs the "attack" by changing the parameters by himself.

Step 1: *Encryption*

Bob chooses a prime number, *[Pb]* > *[M]*, such that he is able to perform a modified encryption of *(M) obtaining a new cryptogram (c1)*:

$$[M]^e \equiv c1 \ (mod\ Na * \ Pb)$$

He sends *(c1)* to Alice.

Important note

The *[Pb]* parameter is the backdoor that allows Bob to return the *[M]* message.

Step 2: *Bob's decryption*

At this point, Bob calculates the *[x]* parameter, which is given by the multiplicative inverse of the function:

$$e * x \equiv 1 \ (mod\ Pb - 1)$$

The result of this inverse multiplication is very important. It gives back *[x]*, a special private key of Bob, which gives Bob access to the decryption of *(c1)*. This is the "magic function" used by Bob to retrieve the *[M]* message.

Bob can now decrypt *(c1)* using *[x]* as his "private key" through the following function:

$$(c1)^x \equiv [M] \ (mod\ Pb)$$

Bob can now mathematically retrieve the *[M]* message from the *(c1)* cyphertext. That goes against what is commonly stated by RSA: it is impossible for the sender to decrypt the cryptogram once it has been performed because the sender doesn't hold the receiver's private key.

Someone could counter that by saying that we have changed the RSA encryption, so we are dealing with a different algorithm and no longer with RSA. That is true to some extent but let's see what happens when Alice decrypts *(c1)*.

Step 3: *Alice's decryption*

Here comes the interesting part of this method: once Alice receives the fake cryptogram, *(c1)*, she is able to decrypt *(c1)* with her private key, *[da]*, and retrieves the secret message, *[M]*, keeping the RSA decryption stage unaltered:

$$(c1)^{da} \equiv [M] \ (mod\ Na)$$

Alice gets the *[M]* message by performing the RSA decryption, unaware of the fake *(c1)* cryptogram received.

As you can see in the following diagram, Bob can perform his self-reverse decryption of *(c1)* "modified cryptogram," while Alice uses her private key to decrypt the fake *(c1)* cryptogram:

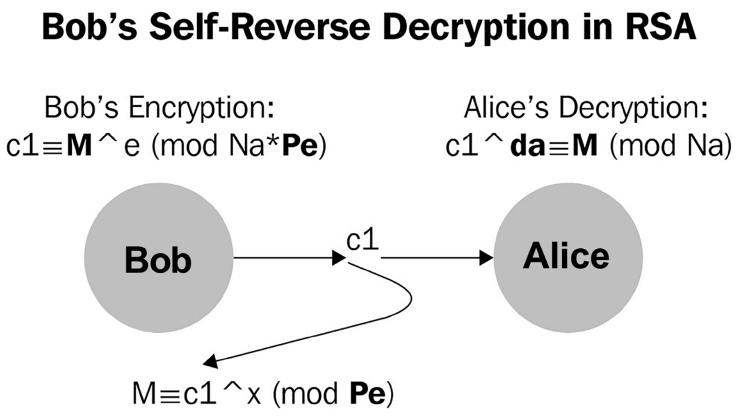


Figure 6.8: A schema of Bob's self-reverse attack

It's easy to demonstrate that if this attack is performed by Eve (an external attacker) who is able to inject her *[Pe]* parameter (backdoor) in Bob's encryption stage, she can spy on the *[M]* message exchanged between Bob and Alice.

In the following diagram, you can see the schema of the attack (spying) taken from Eve:

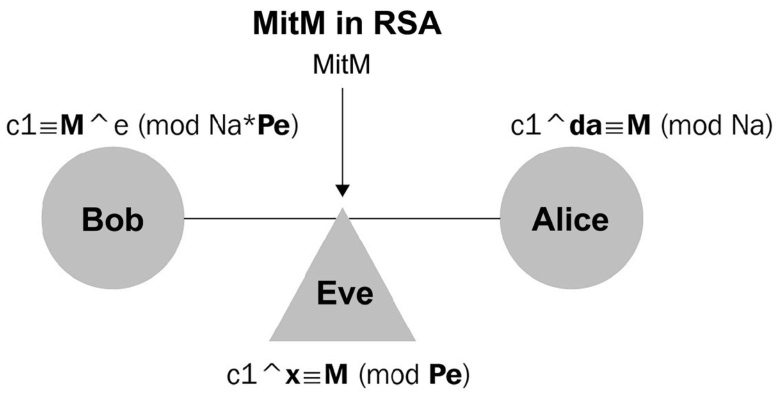


Figure 6.9: A schema of Eve's spying attack (MitM)

With this scheme, not only can Eve retrieve the *[M]* message, but she can also modify it by sending a fake new message, *[M1]*, to Alice, deceiving Bob and Alice, and then recreating a new RSA encryption (original) after having read the message sent by Bob.

To make that possible, Eve has to face off with a problem: the counting of bits in the OpenSSL library. That is because the (fake) encryption stage generates an overflow of bits in *(c1)* that is much larger than the original *(c)* cryptogram. In mathematical terms, *c1 >> c*. This could be a problem if there is a preselected *limit* of bits given by the parameters of the algorithm. This limit could be related to the key length or other parameters in the algorithm, for example, the *N* modulo. In other words, if *N = 2000* bits, then *(c1)* cannot be bigger than *2000* bits; otherwise, *(c1)* is not accepted.

I found a way to avoid this problem by adopting a compression algorithm of my invention, able to generate an output cryptogram in the *(cx) < (Na)* encryption stage. In this way, the cryptogram generated by the fake *(c1)* encryption stage is reduced to *(cx)* and accepted as a *good* parameter by the OpenSSL library, overcoming the problem of the bit-length limit. This compression algorithm can also be used for other scopes: for example, to compress and recompose a cryptogram in transmission to save bandwidth.

As you can see in the following diagram, Eve can spy on the *[M]* message transmitted by Bob and also modify it with *[M1]*, sending to Alice a new cryptogram, *(ce)*, re-generated with an original RSA encryption stage:

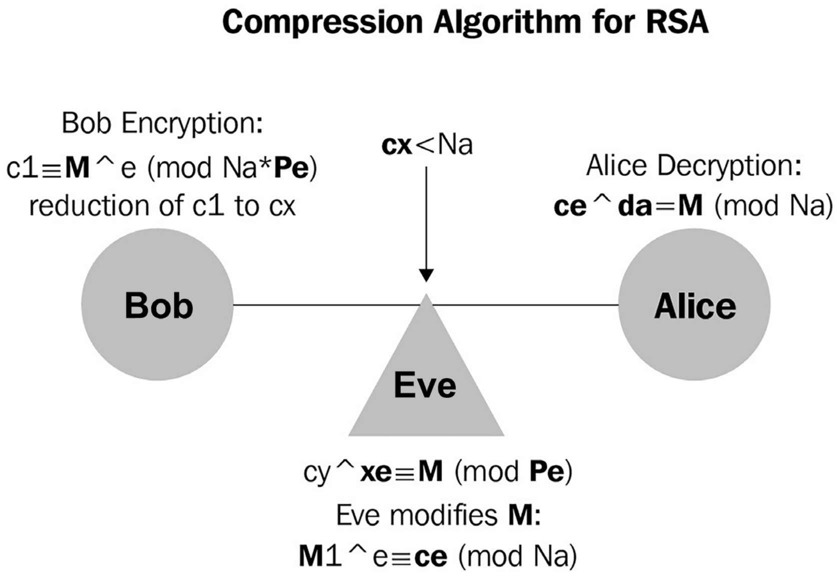


Figure 6.10: Eve attacking through the compression algorithm

Now we should be able to answer another question: if a digital signature is requested from Alice on the *[M]* message, is Eve able to perform this attack?

In other words, is there some way to counter-fight these attacks?

Let's discover the answer in the next section, where I will talk about a new method of digital signatures that is valid not only for RSA but also for most of the private/public key algorithms.

A new protocol to protect RSA and asymmetric algorithms from spying

I will answer the question about whether it is possible to defend RSA against self-reverse decryption, and in general from spying method attacks, by saying that there are multiple ways to avoid these attacks depending on who performs the attack. One of the remediations to avoid these attacks is related to digital signatures (we looked at digital signatures in *Chapter 4*, *Hash Functions and Digital Signatures*). However, in the case of fake encryption and backdoors, as we have seen in the previous section, a special kind of digital signature is required to counter these attacks.

So, I invented this new protocol to guarantee the receiver will decrypt a non-manipulated cryptogram.

This protocol provides a digital signature on the cryptogram made by the transmitter.

Indeed, if Alice (receiver) requests Bob (sender) to digitally sign the cryptogram *(c)* with his private key *[db]*, then Alice will be sure that no one has spied on or modified the *[M]* message and the cryptogram *(c)* was truly signed by Bob and performed with the original RSA encryption.

Let's discover how this protocol works.

Bob signs the cryptogram *(c)* using his private key *[db]*:

$$c^{[db]} \equiv Sb \ (mod \ Nb)$$

Bob sends his signature *(Sb)* to Alice, and Alice can verify whether the following condition has been met:

$$(Sb)^{e} \equiv cv \ (mod \ Nb\ )$$

With this operation, Alice (or everyone) finds the cryptogram through *cv* (cryptogram verification).

Then, if Alice finds out that the message *[M]* elevated to the public parameter (*e*) corresponds to the cryptogram received, so *cv* = *c*, then she can be sure enough that no backdoor will be injected into the algorithm:

$$[M]^{e} \equiv c \ (mod \ Nb)$$

- If *cv = c* then the message is accepted by Alice.
- If *cv is not equal to c* then Alice will refuse the message.

Indeed, performing the operation *cv = c*, Alice compares the cryptogram received with the cryptogram obtained from decrypting the *(Sb)* signature. If it is the same, then no backdoor was injected by a third party. (It is worth noting, of course, that in cryptography nobody can be 100% sure that there is no spy.) If Alice obtains cryptogram *(c1)* or *(cx)*, or another cryptogram instead of *(c)*, she understands something is wrong.

Important note

Here, the signature is performed on the public parameter cryptogram *(c)*, not on the *[M]* message as is normally done. This has been done so the function can be performed by signing (c) directly and not its hash value. In practice it's absolutely recommended to perform a digital signature on *[M]*.

This protocol works for all cryptograms *(c)* regardless of the asymmetric algorithm selected, simply substituting with the private key of the sender the exponent that elevates the cryptogram *(c)*.

In the next section, we will explore the digital signatures on the MBXI algorithm, and we will learn that there are different ways in which to digitally sign this algorithm.

Digital signatures on MBXI

Returning to MBXI, we notice that *[x]*, the reformulated encryption key, is able to perform the encryption:

$$C \equiv M^x \ (mod\ p)$$

[x] results in the inverse of *[y]*, the decryption key, in the following function:

$$C^y \equiv M \ (mod\ p)$$

In mathematical language, the encryption equation looks as follows:

$$\{[K_a^b + e_B]\ (mod\ p) \ * \ x\ \} \equiv \ 1\ (mod\ p-1)$$

This result is the inverse of the decryption equation, *[y]*:

$$y \equiv \{[K_b^a + e_B]\ (mod\ p)\}$$

Let's perform a test with numbers to understand it better:

- $x = 3009$
- $y = 4955$

If we input $x = 3009$ in the inverse function ($mod\ p\text{-}1$), we can find the result [y] using Mathematica:

$$Reduce\ [3009 * x\ ==\ 1, y, Modulus -> p - 1]$$

$$y\ ==\ 4955$$

That means if Bob sends a message using MBXI, he will share a *[secret key]* type with Alice.

Another problem arises: how is it possible to avoid a MiM attack in a symmetric algorithm?

As you can see, MBXI has more characteristics of an asymmetric algorithm than a symmetric algorithm, so let's analyze the algorithms of the digital signature for MBXI.

As we have learned, MBXI could be defined in the same way as D-H, a shared key algorithm. However, in MBXI, in contrast to D-H, it is possible to perform a digital signature; in fact, we will discover that with MBXI there are different modes to digitally sign the message.

For more clarity, I want to recall the scope and functions of the digital signature.

Digital signatures should satisfy the following conditions:

- The receiver can verify the sender's identity (authenticity).
- The sender cannot disclaim the transmission of a message (no repudiation).
- The receiver cannot invent or modify a document signed by someone else (integrity).

Moreover, after you have learned about my backdoor creation, let me add another condition that digital signatures should avoid:

- Anyone not authorized can't look into the document of someone else without being detected (no spying).

This last condition will be one of the principal objects of our study. So, a typical scheme of digital signatures consists of three steps:

1. An algorithm for key generation, *(G)*, that produces a couple of keys, *{(Pk)* and *[Sk]}*, where *(Pk)* is the public key for the verification of the signature and *[Sk]* is the secret key owned by the signatory used to sign the *[M]* message.

2. An algorithm for the signature, *(S)*, that takes the message, *[M]*, as the input (usually the hash of the message) and a secret key, *[Sk]*, to produce a signature, *(s)*.
3. A final algorithm for verification, *(V)*, takes the *[M]* message as input (usually, this is the hash of the message), the public key, *(Pk)*, and a signature, *(s)*, and based on that, it accepts or rejects the signature.

   Given all these conditions and constraints, our digital signatures could be of two types:

   - **A direct signature**: This is a scheme to retrieve the *[M]* message directly from the signature function (*S*). It is commonly used in RSA to perform the digital signature (*S*) on the message, as we have seen in *Chapter 4*, *Hash Functions and Digital Signatures*:

     $$[M]^d \equiv S\ (mod\ N)$$

     And its correspondent verification stage is $S^e \equiv M\ (mod\ N)$.
   - **A signature with an appendix**: It doesn't need the original message to verify its validity (an example is ElGamal's signature, in which the *[M]* message doesn't need to be signed directly to verify the signature).

Now that we have understood the concepts of digital signature methods, we can dive deeper to analyze any single case applied to MBXI.

A direct signature method in MBXI

The direct signature is a method that directly involves the message *[M]* signature. However, as we have seen in *Chapter 4*, *Hash Functions and Digital Signatures*, in some cases, it is much better to apply the hash of the message *H(m)*; otherwise, it could be possible to recover the original message *[M]* from *(S)*.

Let's see how this method works in MBXI. From the MBXI algorithm, we have the following parameters:

- Alice's private key: *[a]* (SK_a)
- Alice's public key: (K_A) (PK_a)
- Bob's private key: *[b]* (SK_b)
- Bob's public key: (K_B) (PK_b)

Bob's public key is given by the following function:

$$K_B \equiv g^b (mod\ p)$$

Assuming Bob sends a message, *[M]*, he will choose a parameter, *[e_B]*, known by Alice and Bob only. That's because, as we said, the function of encryption is verified only under determinate values of *[e_B]*:

$$(E): \{[K_a^b + e_B]]\ (mod\ p)\} \ast x \equiv 1\, mod\ (p-1)$$

Bob and Alice convene to determine *[e_B]* autonomously, in the sense that *[e_B]* is generated step by step, through a process that I have called *joint iteration*. This process consists of a progressive iteration of *[e_B]* until the functions of encryption results are verified.

So, we can reformulate (as mentioned earlier) the preceding encryption equation into the following mode:

$$C \equiv M^x\ (mod\ p)$$

Here, *[x]* is the result of the preceding functions of encryption.

If *[M]* is the message sent in the session and *H(m)* is its corresponding hash value, Bob can sign *H(m)* in the same way he performed the encryption:

$$S \equiv H(m)^x (mod\ p)$$

Here, *(S)* is the digital signature.

When Alice receives the cryptogram *(C)*, she will also receive the signature *(S)* calculated with the hash of the corresponding *[M]* message.

Alice can decrypt the signature *(S)* with the same equation that was used to decrypt *[M]* but with a different *[e_B]*. As we have discovered, Bob has iterated *[e_B]*, bringing its value into the next step of verification of his encryption function:

$$H(m) \equiv Sy\ (mod\ p)$$

Here, *[y]* is given by the following decryption function:

$$y = \{[K_b^a + e_B]\ (mod\ p)\}$$

As you can see, Alice verifies the signature *(S)* with the public parameter of Bob, *(K_B)*, combined with her secret key *[a]*.

Nobody other than Bob can be the sender. An attacker could only use Bob's private key *[b]* in order to perform the following signature equation *(S)*:

$$(S): \{[K_a^b + e_B]\ (mod\ p)\} \ast x \equiv 1\, mod\ (p-1)$$

However, that means an attacker can use the discrete logarithm to return from *(K_B)* Bob's private key *[b]*. But as we have seen, that is a very hard problem now.

Indeed, remember that *[x]* is the result of the preceding function *(S)* and it needs Bob's private key *[b]* to be performed.

But let's examine another method of digital signature to apply to MBXI.

The appendix signature method with MBXI

An alternative method to signing the message with MBXI is called *a signature with an appendix*.

The signature with appendix is a method that we have already found in ElGamal (*Chapter 4*, *Hash Functions and Digital Signatures*). The MBXI algorithm arrives at determinate equality of results, which proves the truth of the signature.

From the MBXI algorithm, we, again, have the following parameters:

- Bob's private key: *[b] (SK_b)*
- Bob's public key: *(K_B) (PK_b)*

These are given by the following function:

$$K_B \equiv g^b \ (mod\ p)$$

The scope of the signature algorithm, *(S)*, is to produce proof, *(s)*, given by Bob such that Alice (the receiver) can verify, *(V)*, its authenticity.

To make this consistent, Bob creates a hash from the secret *[M]* message such that it will not be possible for anyone who doesn't know the original message to recover *[M]* from *H(m)*.

Hence, Bob wants to sign *H(m)* to demonstrate effectively he is who he claims to be.

Step 1: *Key generation* (*G*)

Bob chooses a random number, *[k]*, in the *(Zp)* ring. Then, he calculates *(r)*:

$$r \equiv g^k \ (mod\ p)$$

Step 2: *Digital signature* (*s*)

Bob computes the digital signature *(s)* based on the hash of the *H(m)* message combined with the random key *[k]* and his private key *[b]* as the following:

$$s \equiv H(m) * [k + b] \ (mod\ p - 1)$$

Bob sends the triple *(H(m), s, r)* transmission to Alice.

Step 3: *Verification of the signature* (*V*)

Alice verifies the following two functions *(V)* and *(V1)*:

$$(V)\ g^{s} \equiv V\ (mod\ p)$$

$$(V1)\ r^{H(m)} * Kb^{H(m)} \equiv V1\ (mod\ p)$$

If $\boldsymbol{V} = \boldsymbol{V}1$, Alice accepts Bob's signature. This could be considered a standard method of signing a message *M* through its hash.

Important note

The random key *[k]* has to be kept secret and changed at each digital signature session.

Now that we have examined the different methods of digital signatures in MBXI, let's look at a mathematical demonstration of the digital signature in MBXI.

A mathematical demonstration of the MBXI digital signature algorithm

In this section, I will demonstrate how the digital signatures in MBXI work mathematically.

Indeed, for the properties of power elevation, elevating the generator *(g)* to the equation in *step 2* covered earlier, we have the following:

$$g^{s} \equiv g^{[k+b]^{H(m)}} \equiv (g^{k})^{H(m)} * (g^{b})^{H(m)}\ (mod\ p)$$

From this, we can substitute $\boldsymbol{g^{k}} = \boldsymbol{r}$ and $\boldsymbol{g^{b}} = K_B$ and obtain the following:

$$g^{s} \equiv r^{H(m)} * K_B{}^{H(m)}\ (mod\ p)$$

That is what we wanted to demonstrate.

Now, let's observe what happens with numbers to better understand how these digital signatures work.

If we take, as input, the same parameters of the previous example of encryption, we have the following:

- *p = 7919*
- *g = 7*
- $e_B = 1$

This is Alice's private key, *[a]*:

- *a = 123456*

This is Bob's private key, *[b]*:

- *b = 543210*

The public key of Alice is as follows:

$$K_A \equiv 7^{123456} \ (mod\ 7919) = 7036$$

And the public key of Bob is as follows:

$$K_B \equiv 7^{543210} \ (mod\ 7919) = 4997$$

Recall that the *[M]* message encrypted by Bob was as follows:

$$M = 88$$

First, let's explore the direct signature *(S)*.

Step 1: Bob calculates the hash of the *[M]= 88* message; suppose that the following is the result:

$$Hash(88) = 1305186650$$

Using our well-known Mathematica *Reduce* function, we gain *[x]*, the secret key of encryption:

$$Reduce[(Mod[K_a^b, p] + eB) * x == 1, x, Modulus -> p - 1]$$

$$x = 3009$$

Using *H(m)* combined with *[x]*, Bob calculates the digital signature *(S)*:

$$S \equiv 1305186650^{[3009]}(mod\ 7919) = 7734$$

Bob sends $(\boldsymbol{H}(\boldsymbol{m}), \boldsymbol{S}) = (\mathbf{1305186650}, \mathbf{7734})$ to Alice.

Step 2: Next, Alice can verify whether the *(S)* signature effectively corresponds to Bob.

Elevating the signature to *[y]*, the result must correspond with *H(m) (mod p)*:

$$S^y \equiv H(m)\ (mod\ p)$$

As we have seen, *[y]* is the decryption function given by the public key of Bob, *(K_b)*, elevated to Alice's secret key, *[a]*, added to the *[e_B]* parameter:

$$y \equiv \{[K_b^a + eB]\ (mod\ p)\}$$

Substituting values in the equation, we have:

$$y \equiv \{[4997^{123456} + 1]\ (mod\ 7919)\}$$

$$y = 4955$$

Alice takes *(S)* and elevates it to *[y]*. The result should be the corresponding value of *H(m) (mod p)*:

$$V \equiv 7734^{4955} = 827\ (mod\ 7919)$$

That corresponds to the following:

$$H(m)\ (mod\ p)$$

$$V' \equiv 1305186650\ (mod\ 7919) = 827$$

If *V = V'*, Alice accepts the signature.

Indeed, Alice can verify that:

$$V = 827 = V'$$

As I have explained in this section, there is another way to sign and verify a signature with MBXI.

Let's view an example using numbers from the signature to form an appendix.

All the parameters (modulo, private and public keys, generator, and e_B) remain the same as before.

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

$$k = 1529$$

Bob calculates *(r)*:

$$r \equiv g^k\ (mod\ p)$$

$$r \equiv 7^{1529}\ (mod\ 7919) = 4551$$

Now Bob is able to perform the *(s)* appendix signature:

$$H(m) = 827 \ (mod \ p)$$

$$s \equiv H(m) * [k + b] \ (mod \ p - 1)$$

$$s \equiv 827 * [1529 + 543210] \ (mod \ 7919 - 1) = 4543$$

Bob sends *(H(m), s) = (827, 4543)* to Alice.

Note that, here, to work with low numbers, I have calculated the modulo (*p*) of *H(m)*.

Alice verifies the following:

- $\mathrm{V} \equiv \mathrm{r}^{\mathrm{H(m)}} * \mathrm{Kb}^{\mathrm{H(m)}} \ (\mathrm{mod\ p})$
- $\mathrm{V} \equiv 4551^{827} * 4997^{827} \ (\mathrm{mod\ p}) = 7147$
- $\mathrm{V}' \equiv \mathrm{g}^{\mathrm{s}} \ (\mathrm{mod\ p})$
- $\mathrm{V}' \equiv 7^{4543} \ (\mathrm{mod\ } 7919) = 7147$
- $\mathrm{V} = \mathrm{V}'$

So, Alice accepts the *(s)* signature.

As we have completed the discussion about digital signatures in the MBXI algorithm, I want to present the evolution of the two algorithms MB09 and MBXI, in the particular environment of the blockchain.

The evolution of MB09 and MBXI: an introduction to MBXX

In 2020, I developed and patented another protocol that involves both MB09 and MBXI algorithms.

In my mind, one of the problems not wholly solved in Satoshi Nakamoto's paper was the *consensus problem*. Another issue (also noticed in MB09) is that we are dealing with a centralized system.

I wanted to overcome these problems, so I needed to implement a scheme such that the following conditions are met:

- The protocol runs in a decentralized model.
- The consensus for the validity of transactions is given by a mathematical deterministic function and not by a statistical probability of attack.

In other words, the problem of the *double-spending* of digital money has to be solved in a cryptographic way, instead of with a *consensus* based on "game theory." Indeed, the consensus problem that Satoshi Nakamoto chose is a method based on the theory of the Byzantine generals problem.

This problem, explained through an informal description, involves a group of generals that obey a superior's order to attack an enemy. If the number of generals (generals = nodes) is more than three and some of them are dishonest, rebelling against the king's decision to attack, the attack could fail. In the following diagram, you can see the difference between a coordinated and an uncoordinated attack:

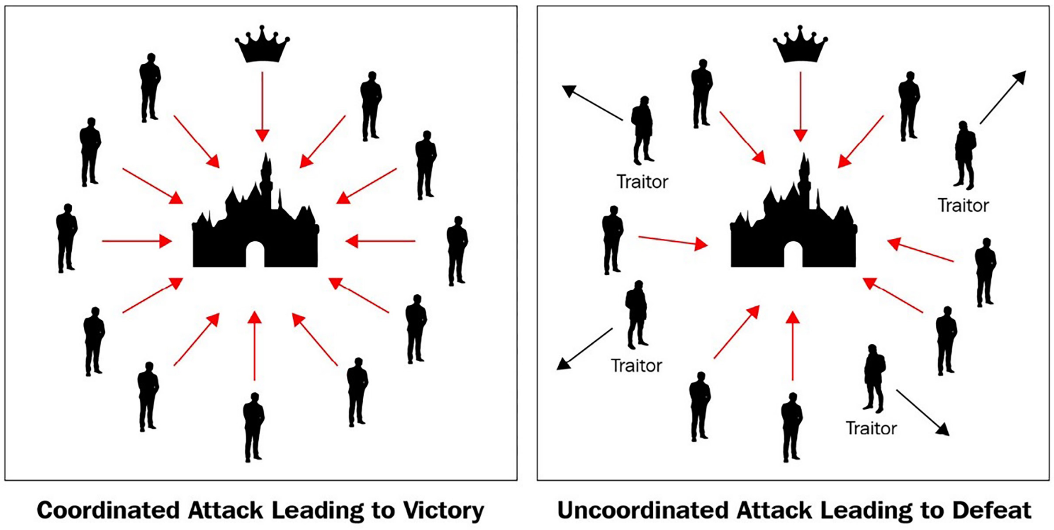

Traitor
Traitor
Traitor
Traitor
Coordinated Attack Leading to Victory
Uncoordinated Attack Leading to Defeat

Figure 6.11: The Byzantine generals problem

Similarly, Satoshi Nakamoto relied on this theory when he supposed that the nodes of a system (the Bitcoin network), deputed to control the double-spending and the truthfulness of transactions, based on the *consensus problem*, could verify the truthfulness of the transactions. Satoshi Nakamoto supposed that the system could support an attack in which the *generals* would, *for the most part*, be honest.

Not everyone knows that in Bitcoin (as in most other cryptocurrencies, too), the deputed miners (through the *proof of work* to the validation of transactions) can't support an attack if the number of fidelity nodes is less than two-thirds of the total number of nodes involved. So, it's not enough that most of the nodes are trustable (51%), but the system requires, at the very least, 75% of the nodes to be trustable in order for them to be reliable.

It's rather strange that Satoshi Nakamoto, after having projected a very sophisticated protocol, mostly based on cryptography, to generate, spend, and give value to Bitcoin, switched to such a non-deterministic method to determine the *consensus* for the transactions.

Indeed, the *consensus* problem of Satoshi Nakamoto doesn't give mathematical evidence of a deterministic result. As I have said, it could only be valid if about 75% of the nodes inside of the system are trustable. Furthermore, if an attack occurs, the system will crash, and all the transactions will be invalid.

Important note

Proof of work is a method invented by Satoshi Nakamoto to produce and transact Bitcoins. It consists of solving a challenge based on SHA-256 (we have already explored the SHA family of algorithms in *Chapter 4*, *Hash Functions and Digital Signatures*).

To avoid the problem of double-spending and maintain anonymous, peer-to-peer transactions, we have to rely on mathematical proof that gives deterministic validity. I imagined a system in which a private blockchain can be governed and controlled only by computers. For example, an *decentralized anonymous organization* is almost completely managed by computers.

This concept of a **Decentralized Autonomous Organization** (**DAO**) was unknown until 2016, or at least it wasn't used yet. I use the concept of DAO as an organization that is self-regulated by a computer network or grid computing with high computation capacity, which is able to perform transactions and run programs to pursue a goal or multiple goals guided only by algorithms. Eventually, this is supported by artificial intelligence to make some decisions. The purpose is to create a decentralized organization, administrated by computers and not by humans. Or better, humans' role in such an organization could be to simply encode the software through the so-called *smart contracts*. To avoid non-determinism, note that artificial intelligence would be used for decisions, not for consensus.

It doesn't matter who owns or is in charge of the hardware's maintenance. That is because this organization could be defined as a meta-infrastructure with no headquarters, no board of directors, no limited time to survive, and, perhaps in the future, no human interference either. That's because hardware machines such as virtual machines exist in the cloud and run their algorithms in a virtual place where time and space don't have any real connotations. If we leave out the problems linked to such an organization's liability and the philosophical concept of creating an entity that is able to live eternally and is unstoppable, what remains is pure mathematics and logic.

Theoretically, this scheme could be unstoppable because, if it's well developed, this organization could be split among a virtual internet organization that is completely agnostic to human power and regulated by algorithms.

Now, I will introduce this protocol and will give an overview of it. In the following diagram, I have represented a decentralized scheme connected through some large central star nodes:

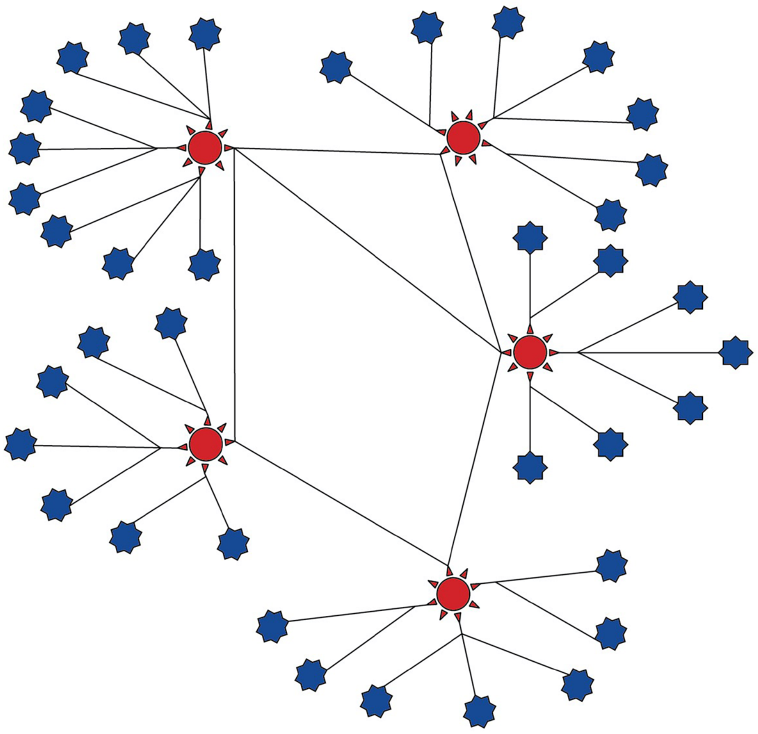


Figure 6.12: A decentralized architecture

Now it's time to go through the steps of the MBXX protocol for digital money transactions in greater detail.

An explanation of the MBXX protocol

The first stage of this protocol is the initialization phase of all the parameters. It represents the *first-level* conditions.

The scheme consists of three steps, starting with the initialization stage of the algorithms adopted, parameters, and keys.

Initialization stage

All the parameters of the protocol are initialized. These parameters are as follows:

- The algorithms that are used in this protocol, such as *(M/alg)*, *(T/alg)*, and *(ZK/Proof)*, for instance, SHA-256, MB09, RSA, D-H, MBXI, ZK13, zk-SNARK, and more.
- An algorithm that has been selected for the digital signature (this mostly depends on the encryption algorithm used).
- The parameters that are used in the algorithm, such as the private keys of the users, *[a], [b], [c]....[n]*, and the correspondent public keys, *(A, B, C, ..., N)*.
- The amounts of digital money detected by each account of the users from where the private keys have been derived.
- The generator, *(g)*.
- The random numbers to generate ZK protocols, *[k1, k2, ..., kn]*.

After this stage, we proceed with delineating the protocol.

Step 1: *Check the total balance with a meta-algorithm*

A first-level algorithm or a *meta-algorithm*, represented by the (*M/alg*) notation, is deputed to check the initial balance and the balances generated after the transactions.

It is not necessary to use the system described in MB09 to perform the homomorphic balance; in fact, there could be something better than this. However, here, I will use the correspondence used in MB09 as a mere representation.

So, let's return to the extended Fermat's Last Theorem, as we know from MB09:

$$[a]^p + [b]^p + [c]^p + \ldots + [n]^p \equiv [z]^p \ (mod\ p)$$

$$A + B + C + \ldots + N = Z$$

This is (*M/alg*). It is made up of both the preceding equations and its role is double:

- It is used to check for the isomorphic balance *(Z)* in a certain instance of time.
- It is used to re-balance the system if new digital money is injected into the system (that is, the sum of the amount of digital money expressed in isomorphic balance during a certain time, *t0, t1, ..., tn*, changes the amount because of the initial balance (*Z0*) augmented if new digital money is injected).

In other words, the *isomorphic calculation* represents each term of the first preceding equation, *[a^p, b^p, ..., n^p]*, transposed as being correspondent with *(A, B, ..., N)*.

So, as we have already discovered, it will be very difficult for an attacker to recover *[a]* even if they know its corresponding public value, *(A)*. That is because this is a *one-directional* function. In this way, the balance of the accounts will be protected with an appropriate padding process. Additionally, you can decide to use another (*M/alg*) notation to obtain this correspondence, such as a *hash system*, where *[a]* corresponds to *(A)---> H[a]*, or another checksum system.

Here, you can discover that *[a, b, c, ..., n]* are the amounts of digital money at a determinate time: *t0, t1, t2, ..., tn*. Each of these elements corresponds to the amount of money belonging to a person or a computer, such as Alice, Bob, Carl, and Nate. Corresponding to any private parameters (the amounts of digital money), *[a], [b], ..., [n]*, we have each public correspondent parameter: *(A), (B), ..., (N)*. These public parameters represent the *isomorphic values* transposed by the hidden parameters in a public domain.

In other words, *[a]* corresponds to *(A), [b]* to *(B), ..., [n]* to *(N)*.

Step 2: The transaction process

Transactions among users take place at any point in time: *t0, t1, ..., tn*. You can figure out that here, *[M]* is a set digital amount of money, where *[Ma, Mb, ..., Mn]* are single amounts of digital money exchanged between the network participants at a particular time, *t0, t1, ..., tn*. For instance, we can think of *[a0] = $10,000* as the initial balance of (*A*) and [*Ma*] = *$1,500* as an amount of money transferred from the account of (*A*) to account *(B)*.

The exchanging of *[Ma], [Mb], ..., [Mn]* between users comes across as a cryptographic algorithm that I have called *Transmission/Algorithm* (*T/alg*), which is made by a private/public key algorithm similar to MBXI.

So, after the first transaction performed, for instance, between *(A)* and *(B)* in times *(t0)* and *(t1)*, the amounts of the single *[a0]* and *[b0]* accounts will be changed to *[a1]* and *[b1]*. For instance, if *(A)* transfers *[Ma] = $1,500* to *(B)* and the initial balance of *A = $10,000*, then the single balance of *(A)* at time *(t1)* will be as follows:

$$(t0)\ [a0] \ = \ 10{,}000$$

$$(t1)\ [a1] \ = \ a0 - Ma$$

$$(t1)\ [a1] \ = \ 10{,}000 \ - \ 1{,}500 \ = \ \$8{,}500$$

A digital signature has to be performed in (*T/alg*) in order to identify the sender to the receiver.

So, in this case, using MBXI, Alice adds her digital signature in this way:

$$Sa \equiv (Hma)^x \ (mod\ p)$$

So, (Sa) is the digital signature added by Alice and (Sa) is given applying the hash of the message *(Hma)* elevated to the encryption key *[x]*, where *(Hma)* is given by the following function: *(Hma) = Hash of [M]*.

And, as you remember from MBXI's signature, the private key *[x]* is given by solving the following equation:

$$\{[K_a^b + e_B B]\ (mod\ p)\} * x \equiv 1\ mod\ (p-1)$$

Here, the (Sa) signature transmitted along with the cryptogram *(c)* will be verified by *(B)* in the opposite way:

$$(Sa)^y = (Hma)\ (mod\ p)$$

After Bob has verified that (*Hma*) corresponds to *[M]*, he accepts the signature and receives the payment.

Step 3: *The verification process*

This step gets all the transactions verified by the administrator of the system. All of this process could be implemented by an autonomous organization similar to a DAO.

The system will operate the first validation on the *homomorphic balance* to check whether there is double-spending among the calculations. If this verification is okay, then after the transactions (made peer to peer through *T/alg*), the admin (or, better, the admins, because the system is computer decentralized) will proceed with a *blind check* of the accounts of the transmitter and the receiver to verify that the balance of the accounts will cover the amount of money transmitted. This verification will be done by applying a particular zero-knowledge protocol. I have called this algorithm *ZK/Proof*.

This ZK/Proof is able to detect whether a user tries to trick the system by sending a fake verification parameter in order to try to set up double-spending. In this case, for example, a user could try to make a payment, but their balance doesn't cover the amount processed. This particular ZK/Proof recognizes the problem automatically and refuses the transaction, without any method of consensus based on a proof of work or other proofs to demonstrate. You can see a scheme of the first transaction and balance performed at time (*t0-t1*) with the MBXX protocol in the following diagram:

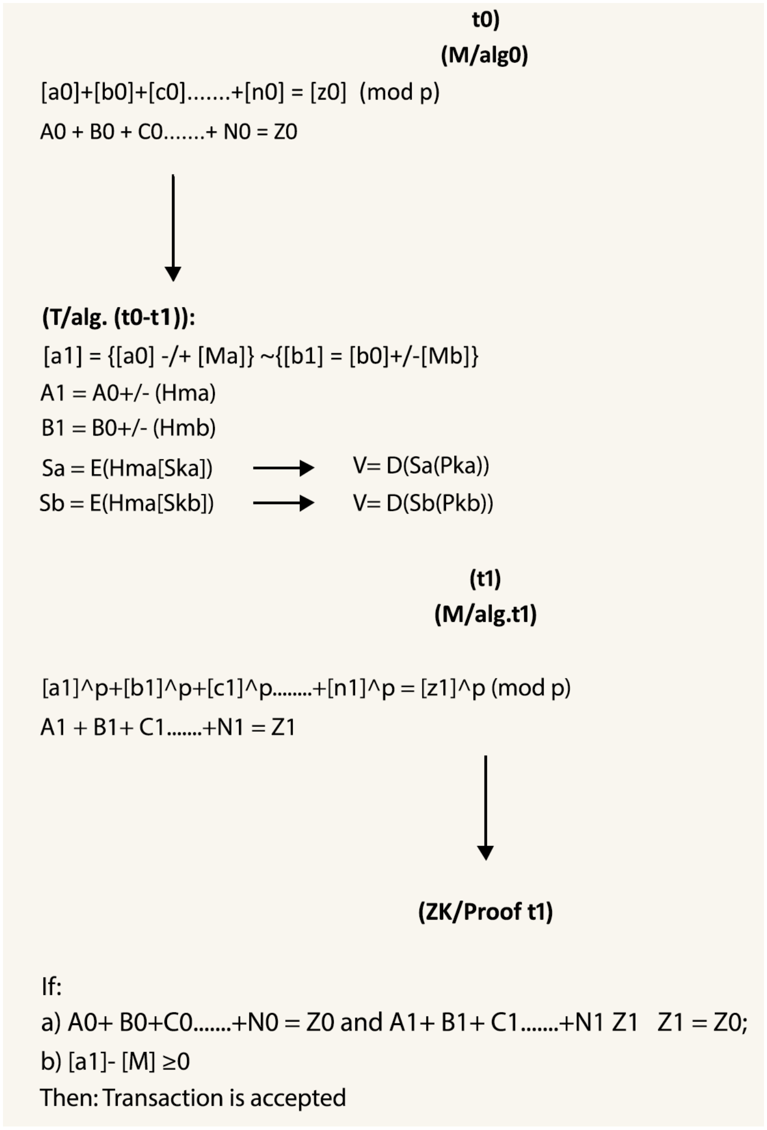

t0)
(M/alg0)
[a0]+[b0]+[c0].......+[n0] = [z0] (mod p)
A0 + B0 + C0.......+ N0 = Z0
(T/alg. (t0-t1)):
[a1] = {[a0] -/+ [Ma]} ~{[b1] = [b0]+/-[Mb]}
A1 = A0+/- (Hma)
B1 = B0+/- (Hmb)
Sa = E(Hma[Ska])
V= D(Sa(Pka))
Sb = E(Hma[Skb])
V= D(Sb(Pkb))
(t1)
(M/alg.t1)
[a1]^p+[b1]^p+[c1]^p........+[n1]^p = [z1]^p (mod p)
A1 + B1+ C1.......+N1 = Z1
(ZK/Proof t1)
If:
a) A0+ B0+C0.......+N0 = Z0 and A1+ B1+ C1.......+N1 Z1 Z1 = Z0;
b) [a1]- [M] ≥0
Then: Transaction is accepted

Figure 6.13: The scheme of the first stage of the MBXX protocol

If no other transactions are performed in *(t1)*, then the protocol will go on with times *(t2)*, *(t3)*, and so on, given by a timestamp.

Notes on the MBXX protocol

If you're curious about this protocol, this section offers more information, focusing attention on particular functions.

Referring to the preceding scheme, we have the function *(T/alg(t0-t1))*. This function expresses all the operations performed between the users at the time (*t0-t1*).

Important note

Pay attention to the - sign in the (*t0-t1*) brackets. It is not a subtraction, but it means that we are moving from *t0* to *t1*.

So, the function is as follows:

$$[a1] = \{[a0] -/+ [Ma]\} \sim \{[\,b1] = [b0] +/-[Mb]\}$$

$$A1 = A0 - (Hma)\ or\ A0 + (Hmb)$$

$$B1 = B0 - (Hmb)\ or\ B0 + (Hma)$$

So, the secret amount, *[a1]*, is generated by *[a0]* added or subtracted by amount *[Ma]*, corresponding to amount *[b1]*, generated by the transferred/received amount, *[Mb]*. Following this operation, the corresponding public parameter, *(A1)*, is generated by the previous *(A0)* parameter subtracted by (*Hma*) if amount *[M]* is received by *(B)*. The same goes for *(B1)*, which is generated by the previous *(B0)* parameter added to or subtracted by the hash of the original amount transferred or received by *(A)*.

This *(Z0)* scheme represents the amount of money in circulation in the system at a given instant, *(t0)*. It doesn't change if no digital money is injected into the system. Instead, if a transaction takes place, it will change the corresponding users' values (parameters) involved in the transaction.

As we discussed in *step 2*, the *transaction process* uses any public/private key algorithm. It's easy to demonstrate that using MBXI, as the system works well. Because the amounts of digital cash are transferred directly from user to user, this can be considered as the effects of a peer-to-peer system, where no third party is involved in the transactions. So, theoretically, no financial or banking institution is required to perform the transactions.

A digital signature (*S*) is added by the sender and verified and accepted by the receiver if the decryption of the (*V*) signature returns the hash of *[M] = (Hm)*. You can see this in the scheme applied to the preceding diagram in the representation of (*T/alg*):

$$Sa = E(Hma[Ska]) -------->\ V = D(Sa(Pka))$$

$$Sb = E(Hmb[Skb]) -------->\ V = D(Sb(Pkb))$$

Here, *(Sa)* is Alice's signature, and *(Sb)* is Bob's signature.

The final step is the verification process, and it's related to a zero-knowledge protocol.

The first verification is as follows:

$$A0 + B0 + C0 + \dots + N0 = Z0$$

$$A1 + B1 + C1 + \dots + N1 = Z1$$

To be valid, the result must be *Z1 = Z0*.

This means that the public parameter, *(Z1)*, which is the result of the *homomorphic balance* related to time *(t1)*, is correspondent to *(Z0)* related to time *(t0)* even if the partial elements of the equation changed in value. This condition guarantees (if no money is injected into the system between times *t0* and *t1*) that the transfer of money between the users into the networks is neutral or balanced.

In other words, the amounts of the transactions made by the users of the system are balanced. However, the verification of this condition by itself does not guarantee that no double-spending has been processed. In fact, it is possible that double-spending was performed if the parties operating in the system are untrustworthy.

Hence, here, we need to implement a *blind verification* or *homomorphic validation* to check that the single amounts of the accounts don't fall below zero. If this condition is also satisfied, then the system (DAO) will ensure that no double-spending has been performed.

Finally, we can say that the necessary condition to satisfy the system is a neutral homomorphic balance. The condition to be validated is that any single balance never falls below zero.

Conclusions on the MBXX protocol

This protocol should overcome the two problems identified at the beginning of this section:

1. The protocol runs in decentralized autonomous mode with no third parties involved in the transactions.
2. The consensus for the validity of transactions is given by a mathematical function and not by a statistical probability of attack.

In terms of the first problem, *decentralized autonomous mode*, I have already appointed the transactions to work through a cryptographic algorithm performed between the users in a peer-to-peer system. Moreover, the verifier (a computer) can only block a transaction if the verification process isn't complete.

The verifier doesn't know how much money has been transferred or the number of single-money accounts. It only knows the original amount of money put into the system and whether a balance is misaligned.

In terms of the second problem, *consensus*, it can be demonstrated that a pure cryptographic model can perform it. This is based on two verifications: the *isomorphic balance* represented by (**M/alg**) and the **zero-knowledge proof** (**ZKProof**) that gives a blind comparison and states whether a single account balance is positive or negative, accepting only positive balances.

For the verification nodes, there is finally the question of whether you would use an open blockchain or a permissioned blockchain. In my view, that is a matter of implementation. Let's not get too far ahead of ourselves: it's a big feat to learn the mathematical basis of the protocol and verify the consensus problem!

After having learned about some protocols in public/private key encryption, let's explore an interesting algorithm in symmetric encryption that is related to a new kind of method called Lightweight encryption, particularly used for IoT and streaming telecommunications.

Lightweight encryption

Lightweight encryption Algorithms (**LEAs**, or simply **LEs**) are a novel collection of cryptographic algorithms designed to have a small implementation footprint and very low energy usage. Lightweight encryption targets an extensive range of devices that are resource-constrained, such as IoT end nodes and RFID tags, which can be implemented in hardware and software across a wide range of communication technologies.

The motivation for lightweight encryption is to leverage lower memory, lower computational resources, and lower power supplies in order to deliver secure solutions that can operate across resource-constrained devices. Lightweight cryptography must optimize implementation costs, speed, security, performance, and energy usage for resource-limited devices. The lightweight encryption algorithm has a low implementation footprint, which takes less computational power to process data, especially in **IoT** (**Internet of Things**).

In the modern era of data transmission and storage, IoT is very important for our society's future. IoT is a complex system that allows us to connect cars and devices, manipulate and monitor home systems, and manage big utility corporations' services.

To manage this sensitive, crucial system, it is important to follow the three main rules at the same time:

- Data must run at high-speed data transmission rates.

- The encryption computational load and memory requirements must be minimized to operate on IoT devices that have few resources and often rely on batteries for power.
- Messages transmitted must be safe and protected because they very often carry sensitive data.

In order to be compliant with these three rules, it's necessary to find a kind of encryption that is fast but at the same time secure.

Let's explore a new algorithm in the space that has got my attention and I have worked on reviewing.

Cybpher algorithm

I propose in this section a new algorithm developed by CyberusLabs, a Poland- and US-based company, that was reviewed by me in 2022. Cybpher is a symmetric algorithm based on **OTP (one-time pad)** algorithm principles. Cybpher is faster than other symmetric algorithms because its simple mathematical operations can be processed in a shorter time. It requires very little memory and very few computational cycles. It could also be considered secure because the logical operations were inspired by one of the most robust ciphers: transposition bitwise operations. But as you know now, after having read this book, nothing can be considered 100% secure if based on classical mathematical operations.

A general scheme of Cybpher LE encryption/decryption can be reproduced as follows

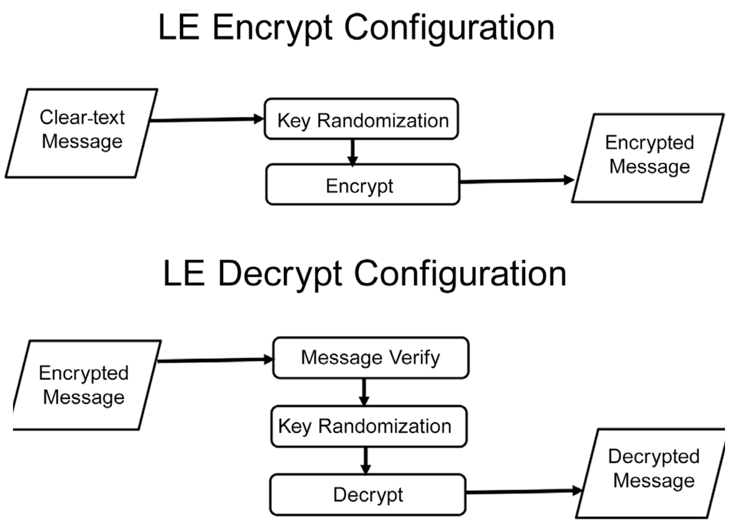


Figure 6.14: This scheme shows the method of encryption and decryption proposed in LE Cybpher

The minimum processor requirement is 8-bit, and the minimum memory requirement is just a little over 2 KB.

The logic behind Cybpher is simple but simple doesn't mean weak. Conversely, cryptography complexity can potentially hide security problems. In a computation analysis done by comparing the principal algorithms of the history of cryptography, we see that algorithms with a high degree of complexity are subjected to more weakness. I remind you that Kerckhoff's principle states: "The weakest part of a chain breaks the entire chain." In other words, complexity is the enemy of security because a longer chain is more likely to be broken. Currently, the most secure, theoretically unbreakable algorithm of conventional cryptography is Vernam. The mathematical operation behind the Vernam algorithm, as we have seen in *Chapter 2*, *Symmetric Encryption Algorithms*, is just one: XOR (Exclusive OR). Just to recap, XOR is a Boolean operator and expresses a sum "bit by bit" between the plaintext and a random key of the same length.

The mathematics of Cybpher is also largely based on Boolean logic operations. But here, there are three differences from the Vernam algorithm:

- Changing the key at every session by autogenerating it
- The ability of this algorithm to generate infinite keys
- Encrypting infinite data starting from one single input key

Indeed, the algorithm relies in large part on a SWAP function to generate the new keys of encryption. The key is a construct of the current transmission and the previous key, which ties both ends of the transmission pair by recording the entire history of transmission between the devices in the current key.

This process connects the two devices of the sender and receiver and creates a mutually authenticated device pair. Therefore, in Cybpher, the active key shared by two or more IoT devices results in an authentication made first to control the exact correspondence with the device selected, then a secure data transmission.

We will analyze the differences between Cybpher and other symmetric algorithms in the next section. For now, we will assert that Cybpher can overcome the problem of a small message being transmitted without using any particular technique of message lengthening. With Cybpher, even if the message sent is very short (as mostly happens in IoT), the encryption works well.

You might be wondering how such a system would manage transmission errors. This is a good question to raise once you're ready to think about the implementation. For the purposes of this book, however, we'll stick to discussing the algorithm from a theoretical point of view.

Let's now demonstrate this algorithm with an encryption stage for a single letter.

Encryption with Cybpher

Suppose we want to encrypt the word "hello" with Cybpher. The algorithm processes an encryption character by character. So, we proceed with a simple example of the first letter, "h." After that, we generate a new encryption key for the letter "e" and so on until the end of the message. I will show the first letter and how to generate the first and the second key, then all the other steps are obviously similar to those.

We know that *h=104* in ASCII code, so we have to encrypt this number.

Step 1: Generate a "key buffer" shuffling 256 numbers (from 0 to 255) and randomizing their position.

Key Buffer (before encryption):

210 50 213 113 21 239 59 11 164 7 158 172 24 176 68 236 80 215 70 153

139 207 42 223 101 136 78 237 147 132 151 150 127 0 232 233 196 133 85

26 154 222 123 227 209 180 145 225 118 117 126 49 192 19 128 74 111 52

37 30 160 187 39 130 69 46 98 200 81 211 23 142 177 212 189 246 114 71

148 106 61 58 131 152 203 115 156 90 185 205 124 83 103 17 29 108 137

182 165 144 161 67 13 251 97 16 62 107 170 228 149 125 173 234 197 143

167 159 169 32 208 250 63 235 243 193 18 253 178 244 195 254 229 84

226 65 140 8 247 79 186 166 241 76 201 206 48 224 94 15 168 55 220 45

174 109 99 60 34 194 102 146 110 135 44 245 240 12 119 217 157 36 216

28 14 238 41 73 93 91 219 255 25 47 54 198 214 202 191 75 1 96 86 9

138 188 4 104 35 33 27 120 181 199 100 155 179 20 230 6 221 88 112 95

92 134 72 31 231 190 183 163 51 57 248 40 66 10 38 89 105 218 171 121

122 2 204 53 87 129 252 43 184 3 56 64 242 141 249 162 77 116 82 175

22 5

Step 2: Pick up a random number x that will be the first key (OTP) among the 256 randomized numbers of the Key Buffer. Suppose it will be *x=210*, so the correspondent position in the Key Buffer is *x1=221*.

Step 3: Do the sum between these two numbers *x+x1 (mod 256)* and generate the following parameter *y: 210+221= 175 (mod 256)*. Find the corresponding value for the *y* position inside another randomized buffer (called the offset buffer), again shuffling 256 numbers. So, we find that the corresponding portion value is *K1 = 6* at position 175 of the Offset Buffer.

Offset Buffer:

85 29 207 24 82 164 118 35 240 249 248 132 205 126 194 138 20 45 4 162

243 250 7 101 163 222 169 106 211 103 25 43 137 80 14 176 33 47 3 144

86 244 54 107 64 152 220 247 56 44 181 16 251 156 1 175 214 23 239 59

149 13 112 121 238 32 134 210 196 135 102 136 198 52 202 161 108 229

19 216 209 232 120 223 110 204 2 34 26 36 245 70 97 140 30 68 119 113

206 60 96 151 21 200 84 124 81 87 109 95 237 218 31 174 153 187 197 65

46 10 203 122 69 127 182 98 183 159 180 67 147 104 221 166 217 235 0

78 129 39 133 226 79 141 72 49 228 66 94 173 9 139 233 208 190 12 252

105 185 99 155 5 179 177 167 40 77 158 227 88 125 117 37 234 55 6 165

212 199 246 157 15 111 253 143 22 38 172 188 62 28 213 8 91 128 131

142 17 186 191 145 83 224 189 93 192 53 11 170 63 115 123 242 130 76

42 74 90 114 184 231 18 148 41 254 73 201 57 178 255 195 61 241 219 58

89 150 27 236 71 146 230 75 154 171 215 116 51 160 225 92 48 193 100

168 50

Step 4: Encryption of the plaintext *h=104*:

Perform XOR between the key *K1* and plaintext *h*:

$$K1 \oplus h = c$$

$$6 \oplus 104 = 110$$

Find the corresponding number at position 110 and substitute 110 with 149 using the Key Buffer. This is the cryptogram of *h=149*:

$$110 --> 149 \; [in\ the\ Key\ Buffer]$$

$$c = 149 \; [Encryption\ of\ letter\ h]$$

Now comes the interesting part: the generation of the *new key session* (valid to encrypt the next letter "e").

From the Key Buffer, pick up the last number:

$$z = 5$$

$$SWAP\ 149 \otimes 5$$

Substitute number 5 with the ciphertext *c = 149*.

This will be the next session key $K2 = 149$.

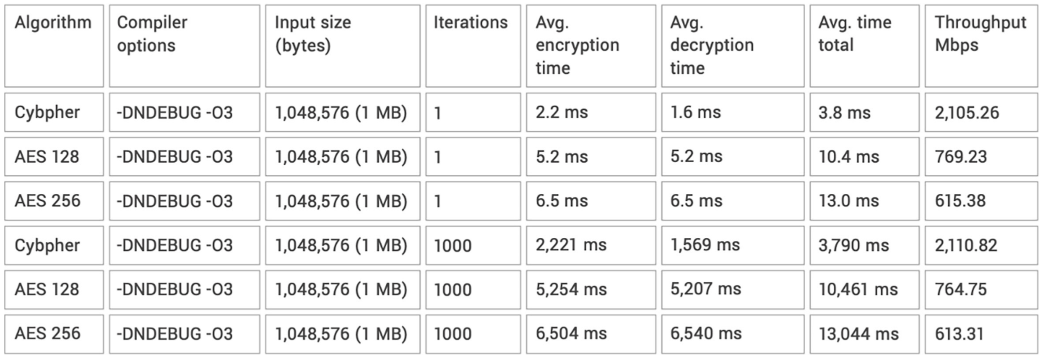
Test and performance

In this section, we will analyze some tests on Cybpher to measure the comparison with other algorithms. In particular, we have chosen to compare the speediness of Cybpher with AES 256, because it is the standard algorithm for symmetric encryption, and AES 128 as it is the standard lightweight symmetric algorithm.

The tests were conducted by CyberusLabs (the owner of Cybpher) on two platforms: an embedded system with a 32-bit ARM Cortex A8 processor running Debian 7.11 and a 64-bit Intel i7 desktop system running Ubuntu 18.04. It is worth noting that, at the time of writing, these tests have not yet been independently verified.

The tests included several different types of input data, different input sizes, and different processing iterations ranging from 1 to 1000.

In the following table, we show the results of the tests on the Intel i7 processor. We are comparing Cybpher with AES128 and AES256 (the standard for symmetric encryption):



Algorithm	Compiler options	Input size (bytes)	Iterations	Avg. encryption time	Avg. decryption time	Avg. time total	Throughput Mbps
Cybpher	-DNDEBUG -O3	1,048,576 (1 MB)	1	2.2 ms	1.6 ms	3.8 ms	2,105.26
AES 128	-DNDEBUG -O3	1,048,576 (1 MB)	1	5.2 ms	5.2 ms	10.4 ms	769.23
AES 256	-DNDEBUG -O3	1,048,576 (1 MB)	1	6.5 ms	6.5 ms	13.0 ms	615.38
Cybpher	-DNDEBUG -O3	1,048,576 (1 MB)	1000	2,221 ms	1,569 ms	3,790 ms	2,110.82
AES 128	-DNDEBUG -O3	1,048,576 (1 MB)	1000	5,254 ms	5,207 ms	10,461 ms	764.75
AES 256	-DNDEBUG -O3	1,048,576 (1 MB)	1000	6,504 ms	6,540 ms	13,044 ms	613.31

Figure 6.15: This image shows the performance times of LE Cybpher compared with AES 128 and AES 256

Important note

Unless stated otherwise, throughput values are in Mbps (Megabits (one million) per second).

We have also measured the entropy of the system. In the following table, we show the entropy related to the encryption with Cybpher:

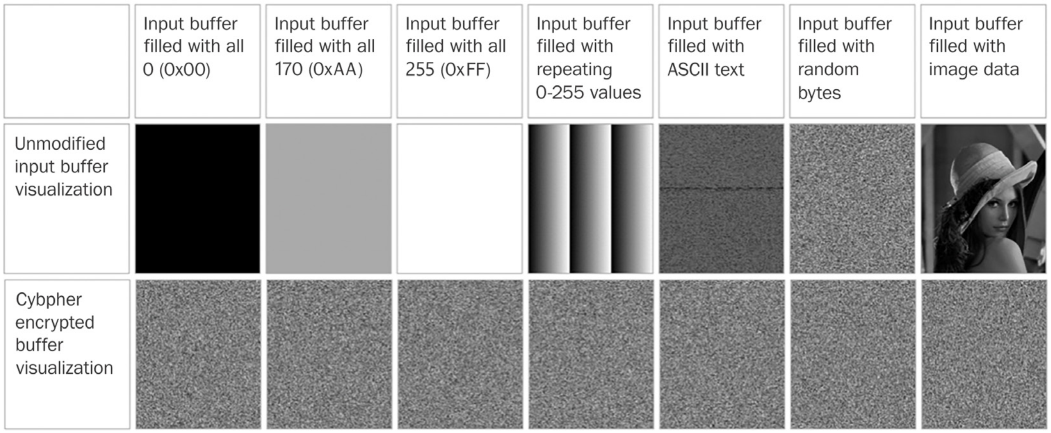

Input buffer filled with all 0 (0x00)
Input buffer filled with all 170 (0xAA)
Input buffer filled with all 255 (0xFF)
Input buffer filled with repeating 0-255 values
Input buffer filled with ASCII text
Input buffer filled with random bytes
Input buffer filled with image data
Unmodified input buffer visualization
Cybpher encrypted buffer visualization

Figure 6.16: This image shows the encryption entropy performed by Cybpher on different data inputs

Lightweight encryption is an emerging type of encryption based on high-speed and lower-power computation. It is mainly used in IoT and other environments where mini processors are required to process and transmit information.

In these sections, I have presented Cybpher, a new kind of LEA that aims to become a standard in the world of LE algorithms.

I have shown the scientific basis of this algorithm, its architecture, and an example of its implementation.

Finally, some tests demonstrated how fast and computationally easy it is.

In this limited demonstration of the algorithm, Cybpher could be considered secure against plaintext attacks and brute-force attacks when the key is truly random, and the key and offset buffers remain well protected. That said, of course, it is new to all of us, and time will tell.

Summary

In this chapter, we analyzed some of my inventions. These algorithms and protocols were primarily invented to solve some challenges related to public/private encryption systems.

Among these protocols, we first saw MB09, which aimed to become a standard for secure digital payment transactions processed in the telecommunication field.

Then, we examined MBXI, a public/private encryption algorithm that was implemented as an alternative to RSA. We explored how it is possible to use a digital signature with this algorithm and the different signature methods: direct and with an appendix.

Moving on, we explored the last protocol: MBXX. This is an evolution of the MB09 and MBXI protocols to overcome the problem of double-spending and the so-called consensus problem. This protocol could be used in the future for a decentralized payment system as an alternative method to proof of work (or proof of stake or other additional proofs) proposed for the validation of transactions in cryptocurrency, based on statistical accuracy.

So, now you have learned about new methods and systems in public/private encryption, such as MB09, MBXI, and MBXX, and you have also become familiar with some schemes in centralized and decentralized crypto systems related to the new era of digital currency.

Finally, I analyzed a new algorithm called Lightweight encryption, a method of encryption valid for IoT telecommunications.

These topics are important because you now have a deep understanding of complex cryptographic schemes. In the following chapters, we will explore many correlations with these topics, particularly in *Chapter 8*, *Homomorphic Encryption and Crypto Search Engine*, where I will explain the basics of the CSE system.

Now that you have learned about these new algorithms, we can explore another fascinating topic: elliptic-curve cryptography.

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, called **secp256k1**. **Elliptic Curve Cryptography** (**ECC**) is highly used in embedded devices because of its smaller keys and reduced power consumption for private key operations. 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 ECC. I will explain what we are doing, but it's worth understanding from the beginning that elliptic curve cryptography is a complex topic, involving geometry, modular mathematics, digital signatures, and logic.

To help demonstrate the mathematics, I will present how secp256k1 is implemented for the digital signature of Bitcoin.

Finally, we will discuss the possibility of an attack on elliptic curves.

In this chapter, we will cover the following topics:

- The genesis of cryptography on elliptic curves
- The 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 about these issues for now; they will become clearer later on in this chapter.

ECC is used as a valid alternative to some of the asymmetric algorithms we have seen in previous chapters, such as **RSA** and **ElGamal**. We will also see that is possible to implement Diffie–Hellman on a particular ECC algorithm.

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, ensuring that transactions are executed successfully.

This chapter intends to take you step by step through discovering the logic behind elliptic curves and transposing it to 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 (see `www.keylength.com`). Such low numbers of bits can be convenient in many implementations that require 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:

$$E: y^2 = x^3 + 73$$

As we can see in the following figure, I have plotted this elliptic curve with WolframAlpha represented in its geometric form:

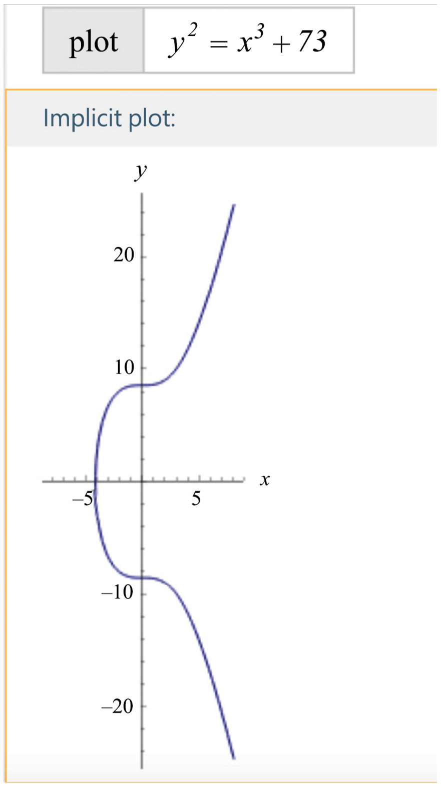


Figure 7.1: Elliptic curve, E: y2 = x3 + 73

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. Here, we define addition on the curve itself.

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:

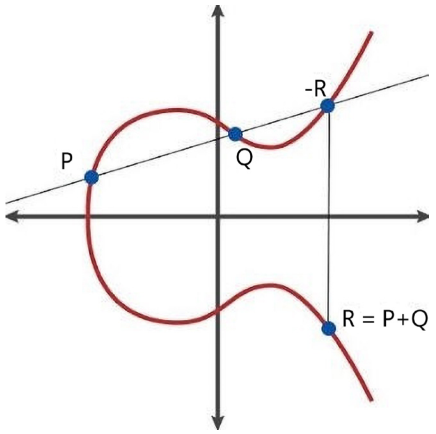


Figure 7.2: Adding and doubling points on the elliptic curve

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:

$$\boldsymbol{s} = (yP - yQ)/(xP - xQ)$$

To compute *xR*, the *x* coordinate of point *R*, we have to perform this operation:

$$\boldsymbol{xR} = s^2 - (xP + xQ)$$

To compute *yR*, the *y* coordinate of point *R*, we have to perform this operation:

$$\boldsymbol{yR} = s\,(xP - xR) - yP$$

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, again finding the reflexed point of the sum, which is the symmetric point *R*, as shown here:

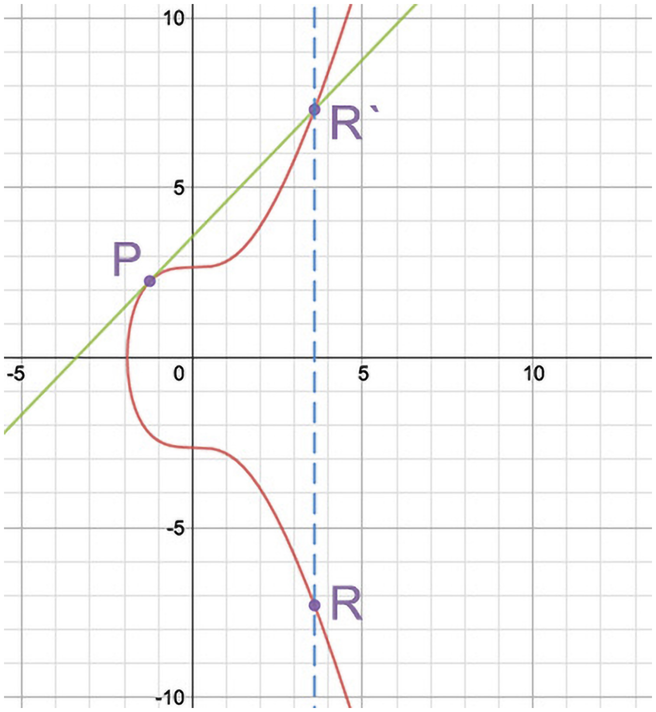


Figure 7.3: 2P point double

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:

$$\boldsymbol{t} = (3XP^2 + a)/2YP$$

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 scenario we need to address when we operate with elliptic curves: how to add vertical points.

Here, *O* is represented by the *point of infinity*, given by the *SUM* value between *P* and *Q* if *xP = xQ*. The straight line obtained by the conjunction of *P* and *Q* is parallel to the *Y* axis, as you can see in the figure below:

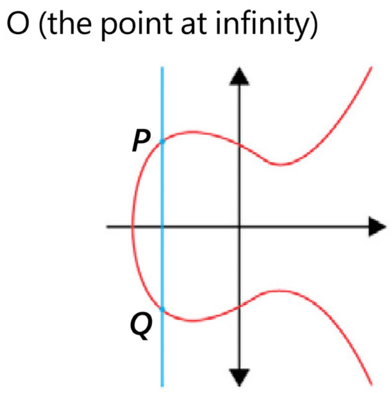


Figure 7.4: Adding vertical points

Algebraically, the point at infinity is given as follows:

$$P + Q = \boldsymbol{O}\ (point\ at\ infinity), if\ xP = xQ$$

Alternatively, it can be given like this:

$$P + P = \boldsymbol{O}\ (point\ at\ infinity), if\ xP = 0$$

As an example, let's assume that we are adding two points, *A* and *B*. They can be defined in the following way:

$$A = (x_A; y_A)$$

$$B = (x_B; y_B)$$

With this in mind, we can understand how to add these points using the following equations:

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

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

If, instead of adding *A* and *B*, we are interested in doubling them*, 2*A = A+A* is obtained like this:

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

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

Now that we have seen how to add and double 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:

$$Q = \{P + P + P + P \dots + P\}\ n\ times$$

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 the double point (geometrically represented in *Figure 7.3*) transposed to 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:

$$2P\ (X_{2P}, Y_{2P})$$

For the same logic, we can go on with *3P*, *4P*, ... *7P*:

$$R = 3P$$

$$R = 2P + P$$

Then, we follow with *4P*:

$$R = 4P$$

$$R = 2P + 2P$$

Then, we follow with *5P*:

$$R = 5P$$

$$R = 2P + 2P + P$$

Then, we follow with *6P*:

$$R = 6P$$

$$R = 2P + 2P + 2P$$

Alternatively, it can be given like this:

$$R = 2(3P)$$

$$R = 2(2P + P)$$

Finally, we have the following:

$$R = 7P$$

$$R = P + \left(P + \left(P + \left(P + \left(P + \left(P + (P)\right)\right)\right)\right)\right)$$

Alternatively, it can be given like this:

$$R = 7P$$

$$R = P + 6P$$

$$R = P + 2(3P)$$

$$R = P + 2(P + 2P)$$

This 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 will 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 7.4*.

If we have *9P*, for example, we will have the following:

$$9P = 2(3P) + 3P$$

$$9P = 2(2P + P) + 2P + P$$

$$9P = P + 2P + 2(2P + P)$$

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 Algorithms*), in cryptography, we are looking for one-way functions. These particular functions make it easy to compute in one direction, but they 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. To this end, *scalar multiplication* is considered to be a one-way function. So, we'll now define the discrete logarithm problem transposed on an elliptic curve.

Let's start with an elliptic curve, *E*. There are some parameters that we can treat as known, namely, *P; Q* is a multiple of *P*. The task is to find *k*, such that *Q = k * P*. This 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 finding *k* means calculating 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?

$$y^2 = x^3 + 9x + 17 \ over\ the\ field\ F_{23}$$

One (naïve) way to find *k* is to compute multiples of *P* until *Q*. The first few multiples of *P* are as follows:

$$P = (16{,}5), 2P = (20{,}20), 3P = (14{,}14), 4P = (19{,}20), 5P = (13{,}10), 6P = (7{,}3), 7P = (8{,}7), 8P = (12{,}17), 9P = (4{,}5)$$

Since *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, which 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 Algorithms*. 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 to elliptic curves too.

First of all, we will deal 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.

> **A cyclic group** means that each point on the curve is generated by a repeated addition (we have seen point addition in the previous section). Remember that we have seen cyclic groups before, when we first learned about D-H in *Chapter 3, Asymmetric Encryption Algorithms*.

- 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*).

A value of *h = 1* is optimal, whereas if *h > 4*, the curve is more vulnerable to attacks (check out the following source to learn more about weak keys: `https://eprint.iacr.org/2005/030.pdf`).

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 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:

$$B = [\boldsymbol{\beta}] * G$$

Alice computes the following:

$$A = [\boldsymbol{\alpha}] * G$$

Now (like in the original D–H algorithm), there is a generation of shared keys exchanging the public parameters:

1. Bob sends *B* (*xB* and *yB*) to Alice.
2. Alice receives *B*.
3. Alice sends *A* (*xA* and *yA*) to Bob.
4. Bob receives *A*.
5. Bob computes the following:

$$K = [\beta] * A$$

6. Alice computes the following:

$$K = [\alpha] * B$$

7. 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, respectively, 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:

$$E: y^2 = x^3 + 2x + 2\ (mod\ 17)$$

Take a look at this *E*, plotted in *Figure 7.5*.

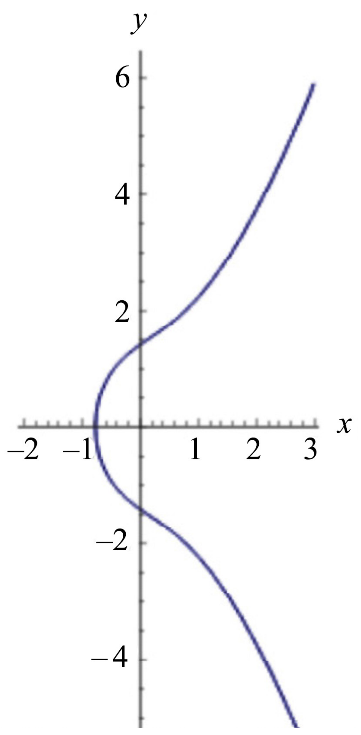


Figure 7.5: The plotted curve used as an example to implement the D–H algorithm

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:

$$t = \left(3X_P{}^2\right)/2Y_P$$

$$t = (3 * 5^2 + 2)/2 * 1 = 77 * 2^{-1} = Reduce\ [2 * x == 77, x, Modulus \rightarrow 17] = 13\ (mod\ 17)$$

$$t = 13$$

Now that we have calculated (t), it's possible to use it to find the x and y coordinates on the *2G* curve:

$$x2G = s^2 - 2xG = 13^2 - 2 * 5 = mod\ [13^2 - 2 * 5, 17] = 6\ (mod\ 17)$$

$$y2G = s(xG - x2G) - yG = 13\ (5 - 6) - 1 = -13 - 1 = -14\ (mod\ 17) = 3\ (mod\ 17)$$

So we have 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:

$$19G = \boldsymbol{O}G$$

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:

$$[\beta] * A = 9A = 9\,(3G) = 8G = (13,7)$$

Bob sends (*B*) to Alice:

$$[\alpha] * B = 3B = 3 * (9G) = 8G = (13,7)$$

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 MitM attacks, just like in the original D–H algorithm (as seen in *Chapter 3*, *Asymmetric Encryption Algorithms*).

Now that we have more confidence in using 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:

$$E: y^2 = x^3 + 7$$

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:

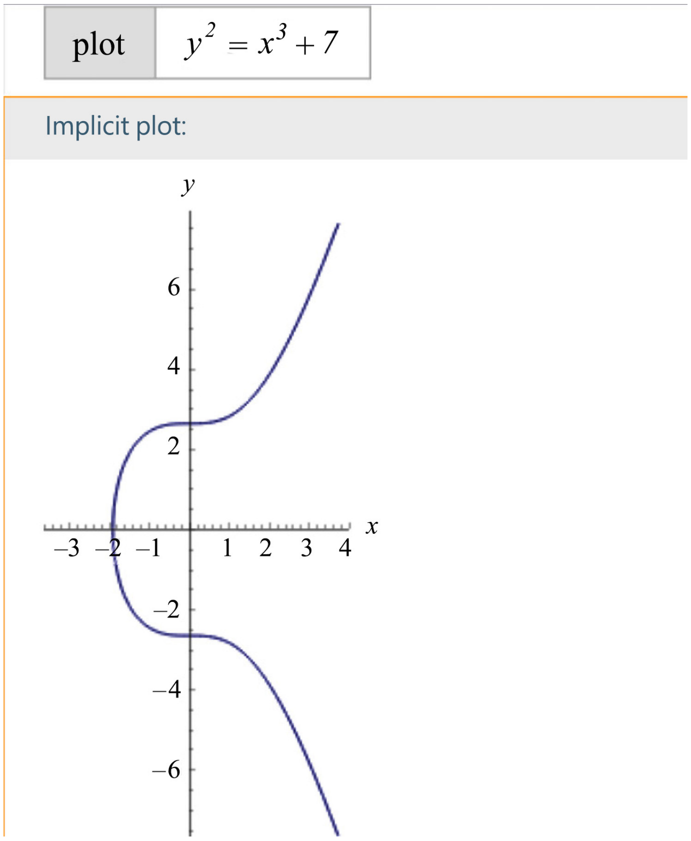


Figure 7.6: secp256k1 elliptic curve

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:

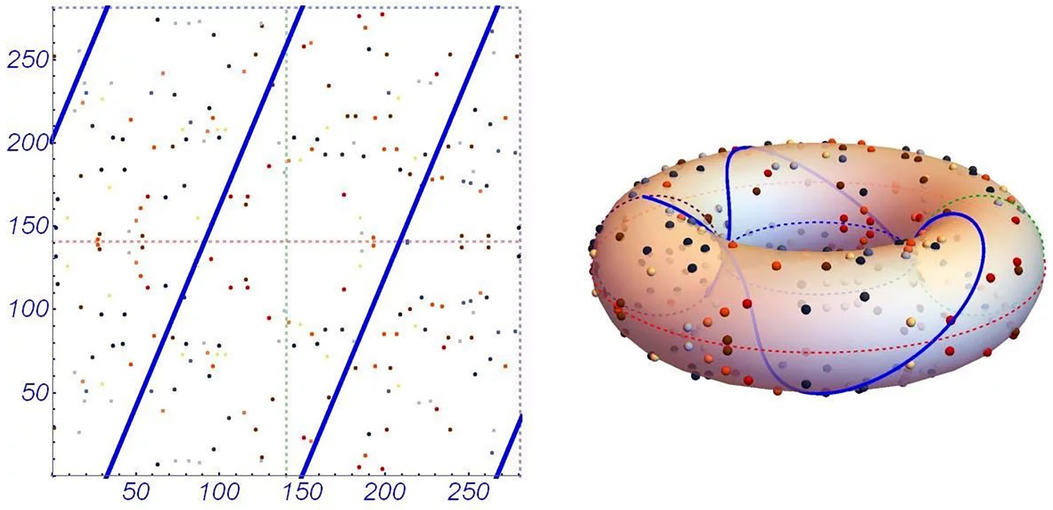


Figure 7.7: 3D representation of an elliptic curve in a finite field

The secp256k1 curve is defined in the *Z* field as follows:

$$Z \; mod \; 2^{256} - 2^{32} - 977$$

Alternatively, written in a different way, this is how it looks as an integer:

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,90

In 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 its counterparts, 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.

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, we will see precisely 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:

$$Q \ \equiv \ [d] * G \ (mod \ p)$$

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*, *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:

1. Alice chooses a random secret, *[k]*, which gives us the following: *[1≤ k ≤ n-1]*.
2. She calculates the coordinates, $R\ (x, y)\ =\ k * G$.
3. Alice finds $r\ \equiv\ x\ (mod\ n)$ (the x coordinate of $R\ (x,y)$).
4. Alice calculates $S\ \equiv\ (z\ +\ r * d)\ /\ k\ (mod\ p)$ (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:

1. Check whether (r) and (S) are both included between 1 and (n-1).
2. Calculate $w\ \equiv\ S^{\ (-1)}\ (mod\ n)$.
3. Calculate $U\ \equiv\ z\ *\ w\ (mod\ n)$.
4. Calculate $V\ \equiv\ r\ *\ w\ (mod\ n)$.
5. Compute the point in secp256k1: $R\ (x, y)\ =\ UG\ +\ VQ$.
6. Verify that $r\ \equiv\ Rx\ (mod\ n)$.

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. Indeed, in the next section, you will find an example that I hope will clarify many aspects of this protocol.

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:

- $p = 67\ (modulo\ p)$
- $G = (2{,}22)$
- $Order\ n = 79$
- $Private\ Key: [d] = 2$

So, 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 use the following formula to calculate the double point:

$$t = (3XP^2 + a)/2YP$$

$$t = (3 * 2^2 + 0)/2 * 22 = 12/\ 44 = Reduce\ [44 * x\ ==\ 12, x, Modulus \rightarrow\ (67)] = 49$$

$$t = 49$$

Calculating *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*:

$$xQ = t^2 - 2XG$$

$$xQ \equiv 49^2 - 2 * 2 = 52\ (mod\ 67)$$

$$xQ = 52$$

In the same way, let's find the *y* coordinate of *yQ*:

$$yQ = t(xG - xQ) - yGx$$

$$yQ \equiv 49(2 - 52) - 22 = 7\ (mod\ 67)$$

$$yQ = 7$$

So the coordinates of the public key (*Q*) are as follows:

$$\boldsymbol{Q}\ (\boldsymbol{x}, \boldsymbol{y}) = (52, 7)$$

Now, Alice can perform her digital signature (*S*).

First, Alice computes the *H[M]= z* hash.

Suppose that *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:

$$\boldsymbol{t1} = (yQ - yG)/(xQ - xG)$$

To compute *xR* (the *x* coordinate of the *R* point), we have to solve this operation:

$$t1 \equiv (7 - 22)/(52 - 2)\ (mod\ 67)$$

$$t1 \equiv 60\ (mod\ 67)$$

We seek *[xR]*:

$$xR \ \equiv \ t1^2 - (xG \ + \ xQ)\ (mod\ 67)$$

$$xR \ \equiv \ 60^2 - (2 + 52) = 62\ (mod\ 67)$$

Let's consider the *x* coordinate of *R*:

$$r = 62\ (mod\ 79) = 62$$

At this point, we can perform the signature by applying the formula:

$$S \equiv (z + r * d)/k\ (mod\ p)$$

$$S \equiv (17 + 62 * 2)/3\ (mod\ 67) = 47$$

We 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:

- $z = 17$
- $n = 79\ (order\ of\ G)$
- $G = (2{,}22)\ Base\ Point$
- $Q = (52{,}7)\ Public\ Key$

First of all, Victor has to check the following:

$$1 \leq (47, 62) \leq (79 - 1)$$

Here we go: the signature passes the first check.

Victor now calculates (w), the inverse of the digital signature (S):

$$w \equiv S^{(-1)}\ (mod\ n)$$

That, as we have already seen on several occasions, means the inverse functions of *S (mod n)* are performed:

$$Reduce\ [(S) * x == 1, x, Modulus \rightarrow (n)] = Reduce\ [(47) * x == 1, x, Modulus \rightarrow (79)] = 37$$

$$w = 37$$

Then, Victor calculates (U):

$$U \equiv z * w\ (mod\ n)$$

$$U \equiv 17 * 37\ (mod\ 79)$$

$$U = 76$$

Now, Victor is able to verify the signature:

$$V \equiv r * w\ (mod\ n)$$

$$V \equiv 62 * 37\ (mod\ 79)$$

$$V = 3$$

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:

$$U * G = 76G$$

$$= 2 * 38G$$

$$= 2 * (2 * 19G)$$

$$= 2 * \big(2(G + 18G)\big)$$

$$= 2 * \Big(2\big(G + 2(9G)\big)\Big)$$

$$= 2\Big(2\big(G + 2(G + 8G)\big)\Big)$$

$$= 2\Bigg(2\Big(G + 2\big(G + 2(4G)\big)\Big)\Bigg)$$

$$= 2\Bigg(2\bigg(G + 2\Big(G + 2\big(2(2G)\big)\Big)\bigg)\Bigg)$$

In 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 brains.

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:

$$UG = 2 * \Big((52{,}7) + 2\big((2{,}22) + (21{,}42)\big)\Big)$$

$$= 2 * \big((52{,}7) + 2(13{,}44)\big)$$

$$= 2\big(2(38{,}26)\big)$$

$$= 2(27{,}40)$$

$$= (62{,}4)$$

$$UG = (62{,}4)$$

In 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 $UG + VQ$ to a unique adding point:

$$R(x, y) = U * G + V * Q$$

$$R(x, y) = (62,4) + (11,20)$$

$$R(x, y) = (62,63)$$

For the last verification, Victor can check the following:

$$r \equiv Rx\ (mod\ n)$$

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 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 that 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 explore 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. This attack worked because Sony didn't properly implement ECDSA, using a static private key instead of a random one.

Attacks on ECDSA 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 used multiple times to sign 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:

$$S \equiv (z + r * d)/k \ (mod\ p)$$

Here, it is presented in numbers:

$$S \equiv (17 + 62 * 2)/3 \ (mod\ 67) = 47$$

$$S = 47$$

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:

$$S1 \equiv (z1 + r * d)/k \ (mod\ p)$$

Using the *Reduce* function of Mathematica, we have the following result for (*S1*):

$$[k * x = (z1 + r * d), x, Modulus \rightarrow (n)] = 49$$

$$S1 = 49$$

where it is given that $(S - S1) / (z - z1) \equiv k \ (mod\ n)$.

Tip

Reduce is a special function that allows us to factorize big modular numbers.

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:

$$Reduce\ [(47 - 49) * x == (17 - 23), x, Modulus \rightarrow (79)]$$

$$k = 3$$

After 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 the denominator to the enumerator on 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, and 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:

$$(47 * 3 - 17)/r = 2$$

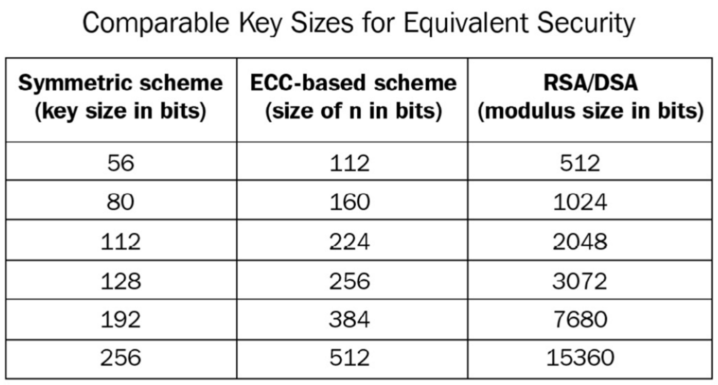
That is the number of the private key, *[d]= 2*, discovered!

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 generating an *S1* signature from (*z*) 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 key strength 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 **Rivest-Shamir-Adleman** (**RSA**) and **Digital Signature Algorithm** (**DSA**). The equivalence you see in the following figure also takes into consideration the symmetric scheme key size (for example, **Advanced Encryption Standard** (**AES**)), for example. This scheme is undoubtedly shorter than the elliptic curve, but since the elliptic curve can spread out digital signatures differently by symmetric scheme encryption, an 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 7.8: 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 an 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. This is a theoretical understanding of the comparative bit sizes between the algorithms.

When you get to the point of considering practical implementations, this should get you thinking about the computational power required by the algorithms.

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 the following in *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 to 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 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 9*, *Quantum Cryptography*. This chapter will 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.

8

Homomorphic Encryption and Crypto Search Engine

This chapter will discuss the technique of searching in encrypted data and provide some basic information on search engines in general and the math behind them. Then, we will discover an innovative search engine that is able to work with encrypted data: **Crypto Search Engine** (**CSE**).

This invention was designed and developed by Cryptolab (`https://www.cryptolab.us/`), which is a cybersecurity company active in data privacy and protection. It was founded in 2012 as a cryptography laboratory by my co-founders and me, who are two data science engineers, Alessandro Passerini and Tiziana Landi. The aim of Cryptolab is to give more privacy and security to the data transmitted and processed over the internet and then stored and shared in the cloud. Another website you can refer to is the web page of CSE (`https://www.cryptolab.cloud/`).

We are going to talk about homomorphic encryption and, in particular, homomorphic searching, which is one of the most fascinating problems I have faced during my career. I am excited to explain what inspired me to start this work and its genesis; moreover, you will discover the math behind search engines and their relative algorithms.

So, in this chapter, we will cover the following topics:

- Introduction to homomorphism and CSE
- The math and logic behind search engines
- Introduction to code theory and graph theory

- CSE explained
- Applications and possible evolutions of CSE

Let's take a deep dive into these topics to learn the secrets behind homomorphic search.

Introduction to CSE: homomorphism

The genesis of CSE dates back to 2014 when I was struggling for several months with a new method of *factorization*. You can understand that the factorization problem and *search in blind* are strictly related to each other. Both these problems, factorization and searching among big data, have similar complexity.

Moreover, both these problems have their domain inside *P=NP*, meaning some problems are *easy* to solve (*P*) while others are *very hard* to solve (*NP*), even if they are supposed to have a very high or infinite level of computation.

Most scientists and data science engineers are convinced that *P≠NP* or all *NP* problems are intrinsically complex and cannot be solved with a polynomial algorithm. I don't think so; I am more interested in finding *solutions* to complicated problems as opposed to saying that solutions don't exist. I am also convinced that there are different ways to obtain a solution. For example, Fermat's Last Theorem has been solved by Andrew Wiles, as was proven through a demonstration of about 100 pages. You can take a look at a formal demonstration at `https://staff.fnwi.uva.nl/a.l.kret/Galoistheorie/wiles.pdf`.

However, I am convinced that it could be solved in just a few pages, and I have some ideas about it. Likewise, suppose we use an efficient quantum processor to perform Shor's algorithm, as we saw in *Chapter 9*, *Quantum Cryptography*. In this case, we can obtain the solution to the factorization of big semiprimes in a few seconds. In the same way, if we use a quantum algorithm, such as Grove's algorithm, to process a search inside a huge database composed of billions and billions of instances (theoretically), we can find the solution quickly.

Coming back to the story of CSE, after we completed *version 1.0* of our *cryptographic library* in 2014 at Cryptolab, I started to implement an algorithm that could search for data (blind). The Crypto library project consisted of amalgamating into one *crypto-library* all the inventions developed by me and my team of researchers between 2009 and 2014. We called this project *Cryptoon* and it turned out to be a C++ library consisting of not only our cryptography algorithms but also symmetric/asymmetric, zero-knowledge, and authentication algorithms, most of which I have already discussed in this book.

At the time, I was enthusiastic about the discoveries made in the field of *homomorphic encryption*. This kind of encryption has a peculiarity: it performs the manipulation of data (operations on it) in an encrypted way and gives back the encrypted mode results. After the results have been achieved, those who detect the decryption key can retrieve the encrypted result by decrypting it and obtaining the plaintext result. Let's make a scheme to better understand this complicated logic.

The operations on encrypted data generate a consequent encrypted result, *E[Z]*, as follows:

Plaintext (A),(B) -------> Encrypted Data E[A], E[B] -------> Operations on Encrypted Data = Encrypted result:

$$E[A]\forall o\ E[B] = E[Z]$$

The decryption of the *E[Z]* result is as follows:

E[Z] -------> Decryption of Encrypted Result [Z] is:

$$D(Z) = (A)\ \forall o(B) = (Z)\ (Plaintext).$$

Note

I adopted a symbol of my invention, ∀o, to denote any mathematical and logical operation (or set of operations) performed between *(A)* and *(B)*. ∀*o* stands for ∀ = for all; *o* = operations.

But don't worry. We will have a better understanding of these concepts later in this chapter. So, let's now introduce the concept of homomorphism.

Homomorphisms are operations performed on *isomorphic elements*. We can find isomorphisms in nature, such as in mathematics. In nature, for example, we can see that the backs of the wings of some butterflies reproduce a concentric pattern in the shape of wings. The skeleton of a person relates to their body, or their shadow, which is a form of isomorphism: elements belonging to a person, an animal, or something else generated, related, or derived from itself.

However, even though we cannot distinguish the face of a person by looking at their shadow, we can certainly recognize some elements of them, such as their shape or curly hair, that identify this individual. We are not talking about similarity, but a relational property between two elements constituting part of the same thing or generated by the same element.

In the following figure, you can see the difference between the two photos: the left one represents isomorphism because the shadow is the reflection of the body on the sand, caused by the sun, while the second does not because the clouds don't derive from the house on the hill (it's just a random event), even if it looks like the shape of the house is drawn by the clouds:

Figure 8.1: Isomorphic example (left) and non-isomorphic example (right)

A bit more formally, we can say that *homomorphic encryption* is a form of *encryption* that allows *computation* on *ciphertexts*, generating an encrypted result that, when decrypted, matches the result of the operations as if they had been performed on *plaintext*.

Let's look at the scheme of homomorphic encryption:

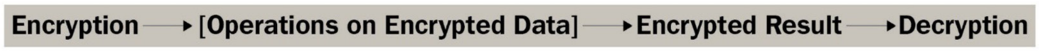


Figure 8.2: Flow of operations in homomorphic encryption

This scheme can be executed by performing mathematical operations, such as multiplication or addition. If the system performs all the mathematical and Boolean operations, in this case, we say that the algorithm is *fully homomorphic*; otherwise, it is *partially homomorphic*. For instance, RSA (Rivest–Shamir–Adleman) or other algorithms, such as *ElGamal* or *MBXI*, are partially homomorphic. In RSA, we can find the homomorphism of multiplication, as we will discover in the next section.

Partial homomorphism in RSA

Let's take the RSA algorithm to explain this correspondence between ciphertext operations and plaintext operations (in this case, multiplication).

We know that RSA's encryption is as follows:

$$[M]^e \equiv c \ (mod\ N)$$

From this, we have the following:

- [*M*] is the message.
- (*e*) is the public parameter of encryption.
- (*N*) is the public key composed by *[p*q]*.
- (*c*) is the cryptogram.

We are supposed to have two messages, *[M1]* and *[M2]*, and we encrypt them using the same public parameters (*e*, *N*) to generate two different cryptograms (*c1*, *c2*). The result of the encryption will be as follows:

$$c1 \equiv M1^e \ (mod\ N)$$

$$c2 \equiv M2^e \ (mod\ N)$$

If we multiply the ciphers (*c1*c2*), we get as a result a third cryptogram (*c3*), such that the following applies:

$$c3 = c1 * c2$$

$$c3 \equiv M1^e \ (mod\ N) * M2^e \ (mod\ N)$$

$$c3 \equiv M1^e * M2^e \ (mod\ N)$$

I have simply substituted the operations on the *[M1]* and *[M2]* encrypted messages with the (*c1*) and (*c2*) cryptograms.

All these operations can be regrouped into one cryptogram, (*c3*), such that we have the following:

$$c3 \equiv c1 * c2 \ (mod\ N)$$

Or, we may even have the following:

$$c3 \equiv (M1 * M2)^e (mod\ N)$$

Hence, we obtained a result of (*c3*), which is the encryption of the multiplication performed on the two cryptograms (*c1*c2*) and, at the *second level*, the operations performed on the messages, *[M1*M2]*. Let's now see what is meant by the *second level*. Decrypting (*c3*) with the *[d]* private key, we obtain the same result as *[M1*M2]*:

$$c3^d \ (mod\ N) \equiv M1 * M2$$

We can verify with a numeric example (because I suppose you could be skeptics) that the following applies:

$$c1 * c2 \equiv (M1 * M2)^{e}(mod\ N)$$

Or, we could also do the following:

$$c3^{d}\ (mod\ N) \equiv M1 * M2$$

This means that if Alice decrypts a cryptogram, (*c3*), with her private key, *[d]*, she gets, with RSA, the result (in linear mathematics) of the multiplication of the messages, *[M1*M2]*, previously performed by Bob and sent by him, hidden, to the unique cryptogram, (*c3*).

This is called *partial homomorphism* of the multiplication in RSA.

Let's look at a numerical example to understand this better.

Let's take some small numbers as an example for verifying RSA partial homomorphism:

- *M1 = 11*
- *M2 = 8*
- *e = 7*
- *p = 13*
- *q = 17*
- *N = 221 [p*q]*

Compute the *c1* and *c2* cryptograms:

$$c1 \equiv 11^{7}\ (mod\ 221) = 54$$

$$c2 \equiv 8^{7}\ (mod\ 221) = 83$$

Now, we calculate (*c3*) in such a way that we consider it to be our *first level*:

$$c3 \equiv c1 * c2\ (mod\ N)$$

We then calculate (*c3*) in another way that we consider to be the *second level*:

$$c3 \equiv [M1 * M2]^{e}\ (mod\ N)$$

So, in terms of numbers, we have the following at the *first level*:

$$c3 \equiv 54 * 83 = 62\ (mod\ 221)$$

We have the following at the *second level*:

$$c3 \equiv [11 * 8]^7 \ (mod\ 221) = 62$$

As you can see, we found a degree of correspondence between the operations (multiplications in this case) performed with the *c1* and *c2* cryptograms and multiplications performed on messages *[M1, M2]*.

Up to this point, it seems that everything is normal. We have only applied simple mathematical substitutions to the equations performed with both cryptograms and messages.

However, another important correspondence turns out to be the decryption of (*c3*), which is what makes the difference. In fact, Alice is able to decrypt the two cryptograms transmitted by Bob (*c1*, *c2*) through her unique private key, *[d]*. Using the well-known *Reduce* function of the Wolfram Mathematica software to find *[d]*, the private key, we have the following:

- *Reduce [e*d= 1, x, modulus -> [13- 1] [17: 1]]*
- *d = 55*

But now, the surprising part is that by decrypting (*c3*), the result of the multiplication of (*c1*c2*), Alice can also ascertain the result of the multiplication of the *[M1*M2]* messages.

Remember that *c3 = 62* and *[d]* = 55, and we have the decryption function in RSA as follows:

$$c^d \equiv M \ (mod\ N)$$

In this case, we have the following:

$$c3^d \equiv M1 * M2 \ (mod\ N)$$

Substituting the numbers gives us the following:

$$62^{55} \equiv 88 \ (mod\ 221)$$

In fact, it turns out that we have the following:

$$M1 * M2 = 11 * 8 = 88$$

This is the result we wanted to obtain!

As you can see, even the RSA algorithm holds some homomorphic properties, including multiplication. However, this is just a case study because we need to perform homomorphic searching differently from how RSA goes about it. So, let's analyze homomorphic encryption a little bit more deeply and its implications for data security and privacy.

Analysis of homomorphic encryption and its implications

Analyzing the partial homomorphic scheme proposed for RSA in the preceding section, we can see an interesting correspondence between the operations on cryptograms (data expressed clearly) and messages (blind data).

This correspondence is just what we call homomorphism.

Now, the problem is that RSA, just like most of the algorithms explored until now, is *partially homomorphic* and can only represent some mathematical operations, such as multiplication or addition. The real difficulty is finding an efficient algorithm that represents all the mathematical and Boolean operations together.

Another simple example (case study) of how a form of homomorphism can be represented is performed by addition.

Let's take *[A]* and *[B]*, two secret values that give *[C]* as their sum:

$$A + B \equiv C \ (mod\ Z)$$

Now, let's take two encrypted value correspondents respective to *A* and *B*.

For example, the encrypted values are *a = 9* and *b = 2*, the sum of which is *c = 11*.

As an example of homomorphism, although this is not technically an encryption, we can generate a homomorphic partial (sum) correspondence such that we have the following:

$$A = 87 \text{——>} a \equiv 9 \ (mod\ 13)$$

$$B = 93 \text{——>} b \equiv 2 \ (mod\ 13)$$

$$C = A + B = 180 \text{——>} c = a + b \equiv 11 \ (mod\ 13)$$

We can also represent these operations in a row, so you probably now have a better idea of what homomorphic means:

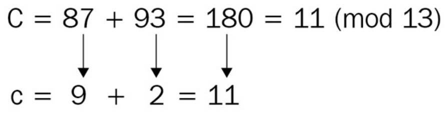


Figure 8.3: Correspondence of the homomorphic sum between plaintext values and encrypted values

Figure 8.4: Searching in blind

As you can see, this simple correspondence makes it possible to operate *blind* with the encrypted values, $a + b = 9 + 2$, and expose the result, $c = 11$, but it can preserve the *privacy* of the numbers, $A = 87$, $B = 93$, and $C = 180$, exposing only an isomorphic value, $c = 11$, which is not the real value of C but represents its corresponding (isomorphic) value. The condition is that the function that transforms the value is a one-way function, so it will be hard to return from (a) to *[A]*, (b) to *[B]*, and the result, (c), to its isomorphic correspondent, *[C]*.

I have deliberately highlighted the term *privacy* (I know it may appear strange to talk about the privacy of a number). It is just that the result we want to obtain while operating with such forms of isomorphism preserves data privacy. Operating with data *blind*, indeed, we obtain a result that can be exposed clearly without the fear that any element of the equation can be discovered.

I started to conceive the project to implement CSE, a search engine able to operate with data *blind*, not *conscious* of what it is searching for, and agnostic of any algorithm used to perform the encryption/decryption of the data retrieved.

Before presenting CSE, I want to analyze a little bit of the math behind search engines (for general purposes and related to CSE) to better understand how search engines are implemented and how fascinating they are.

Now, it's time to explore the math and logic behind traditional search engines and understand how complex it is to perform an efficient search on encrypted data.

Math and logic behind search engines

Everything related to the *network problem* and network theory started from Konigsberg's bridge dilemma in 1735. Euler was the first mathematician who solved a dilemma (relating to an apparently simple question) about crossing a set of bridges over the river Pregel, in the ancient city of Konigsberg in Prussia, known today as Kaliningrad.

As you can see in *Figure 8.5*, seven bridges link the river's two banks (A and B) through two islands (C and D). The question is: *is there any way to cross all seven bridges of Konigsberg by going over them just once?*

You can try to find a solution by yourself by looking at the next image of the bridges:

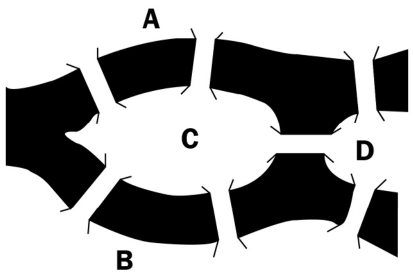


Figure 8.5: The seven bridges of Konigsberg

The key is to recognize that we are facing a network (made of bridges), and we have to draw it to understand the solution. Compared to the seven bridges network, there are only four points where a wayfarer can be located at a specific moment, as we can see in the following diagram:

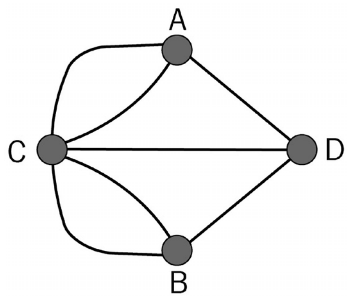


Figure 8.6: The network of Konigsberg's bridges

Euler demonstrated that the question of crossing all seven bridges by going over them just once has no solution. That is because the network of bridges has four hubs (points), corresponding to seven links (bridges), and when the number of hubs is even while the number of links is odd, the problem has no solution.

The issue of crossing networks efficiently is one of the most important elements to consider if you want to design and implement an efficient *search engine* or *social network* or perform *big data analysis*. We have to find a way to retrieve the instance (the searched word, for example) and then cross the network and return with the least effort possible.

A similar problem is represented by searching for the best itinerary among many choices represented, for example, by different roads on a map. Have you ever wondered why Google is a champion not only when it comes to finding any answers to your questions, but also for finding the best solution regarding an itinerary covering hundreds of miles and finding the destination in a few milliseconds?

The problem is related to *circuits*, in particular, *Eulerian circuits* and other similar mathematical problems. To explore this question further, let's examine an example of how a network operates, as illustrated in the following figure, named **Papillon**:

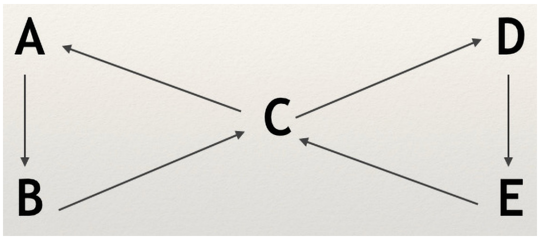


Figure 8.7: Eulerian Papillon network

The Papillon network has a Eulerian circuit of this type:

A—>B—>C—>D—>E—>C—>A

In this representation, as you can see if you experiment by yourself, we pass by C twice (remember that we started at A, but it works the same if we start at C or any other point).

By changing the path a little bit, crossing the network through the same five hubs – A, B, C, D, and E – is possible without passing point C twice, as you see here in *Figure 8.8*:

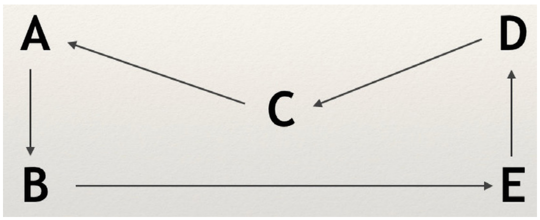


Figure 8.8: Eulerian cycle

In this case, the circuit is *A—>B—>E—>D—>C—>A*.

These problems send us back to the following question: *how do we find the best itinerary among a circuit made of hubs?*

If we want to implement an efficient search engine and ask it to find the best itinerary to cross over a group of hubs (as we have previously represented in *Figures 8.7* and *8.8*), we should impose some conditions. As you know, in real life, sometimes the constraints are expressed by traffic or by avoiding highways or tolls, but most commonly, the first condition is represented by the shortest itinerary of the path. In our case, the first condition is to find the shortest path to complete the cycle among all the points, pass by each point only once, and move in a clockwise direction as the last condition. Our search engine has to calculate the shortest distance starting (for example) from A and returning to A, intersecting all five points just once and moving clockwise. In this case, the representation depicted in *Figure 8.9a* is a solution (but not the only one).

Another solution is *A—>B—>C—>E—>D—>A*, proposed in *Figure 8.9b*. The scheme proposed in *Figure 8.7* (Papillon) doesn't respect the second *rule* we gave to our search engine, so it is not valid.

To find the best solution, we have to assign a value to each segment linked by the lines that represent the itinerary that our search engine will run. Suppose that a segment (for example, *A—>B*) has the following values:

- *A->B = 1*
- *B->E = 3*
- *A->C = B->C = C->E = D->C = 2*

The itinerary's values shown in *Figures 8.9a* and *8.9b* are equally efficient.

Itinerary (IT_1) has a value of $1 + 3 + 1 + 2 + 2 = 9$.

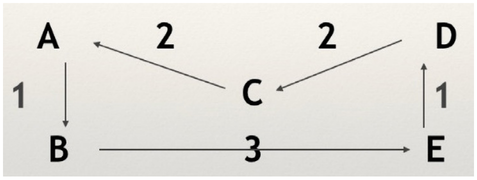


Figure 8.9a: IT_1: a possible solution for the cycle

Our itinerary is equally efficient as itinerary (IT_2), as you can verify in *Figure 8.9b*:

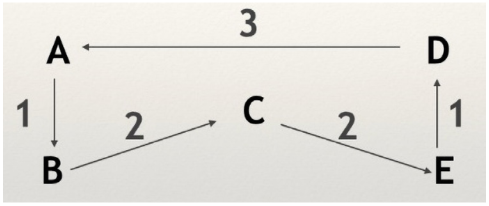


Figure 8.9b: IT_2: another possible solution

Itinerary (IT_2) has the same value as $IT1: 1 + 2 + 2 + 1 + 3 = 9$.

So, the two solutions, $IT1: A \text{—}> B \text{—}> E \text{—}> D \text{—}> C \text{—}> A$ and $IT2: A \text{—}> B \text{—}> C \text{—}> E \text{—}> D \text{—}> A$, have the same total values, IT = 9, which is the most efficient for our search engine.

Now, suppose we want to calculate the total value expressed in the Papillon circuit (*Figure 8.7*). In that case, you will discover that it is less efficient to run the entire circle (the total value of Papillon is $ITp = 10$), but it's the best scheme *to search for* the point (*C*) and return to A in less time and length, as you can see in *Figure 8.10*. In fact, in the Papillon circuit, we can return directly to *A* from *C*: $IT3: A \text{—}> B \text{—}> C \text{—}> A = 5$.

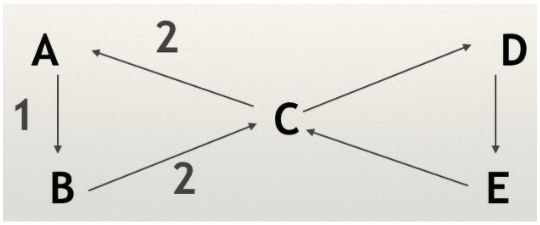


Figure 8.10: The IT_3 itinerary

In these graphs, you start to understand the logic behind a search engine. One of the principal requirements of a search engine is to be efficient. We have to keep in mind that whenever we design a search engine, be it encrypted or not, efficiency is an important element. Just like traveling along the street, the path expressed in miles, or another metric, always has a correspondence expressed in time to process the itinerary. That is the same as when you ask Google to process the best way to reach a destination and choose the travel time as a variable instead of the travel distance.

Time is the primary variable to consider when we launch a query on a search engine. It's agreed that two seconds is the maximum time limit to answer a query (in an unencrypted way). In other words, for example, a user is not prepared to wait more than two seconds for an answer while searching for something on their cellphone. It's for that reason that we have to project a circuit with the maximum grade of efficiency.

One of the constraints adopted when we embarked on the CSE project was to stay below two seconds of elapsed time per query. This seemed impossible, given similar (but not identical) previous encrypted search engine projects, such as TOR: `https://www.torproject.org/`.

When we deal with encrypted data, the latency of the communication becomes heavier. This is because mathematical operations (as we have seen in this book) have more computational power than operations performed on plaintext. Multilayers of encryption (as in TOR) do not always reinforce the system but certainly make it computationally heavier.

Always remember that in cryptography, the weakest link in the chain destroys the entire chain. In the next section, we will examine another essential element in graph theory that is crucial for the implementation of a search engine: tree graphs.

Introduction to trees: graph theory

A **tree graph** is a discrete mathematic structure visualized in a geometrical representation as a tree. Graphs are used in mathematics such as in AI and other fields of applications to make decisions, to represent the best path to reach a destination. As we will see in the next section, the Huffman code can be used in tree graphs to encode text.

A tree graph uses vertices (*v*), also called nodes or points, that are related to each other. Each of the related pairs of nodes is called an edge (or link or line) and two nodes are connected by only one path.

In the following figure, you can see a tree where 1 is the root of the tree, 2 to 7 are the nodes, and 8 to 15 are the **leaves**. The segments that link each of the points are the edges.

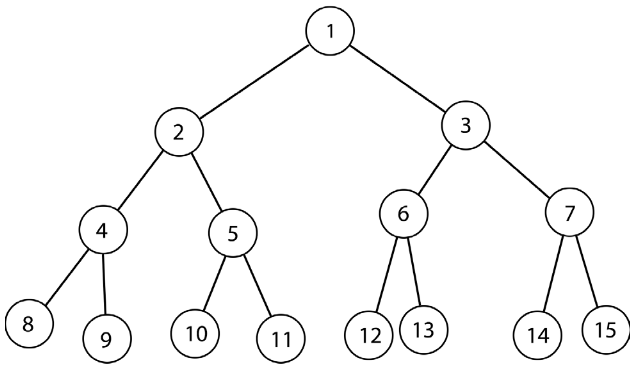


Figure 8.11: A tree graph

We have seen many cryptographic algorithms in this book. The main scope of these algorithms is related to data security. However, the main scope of the algorithms we are now approaching, also called *codes*, not only concerns security but also aims to preserve the information in a synthetic and usable way. This concept is related to retrieving information in the most efficient way possible. Like the problem of the bridges of Konigsberg and Eulerian circuits, tree graphs can be used to calculate the best way to retrieve information.

This problem also goes by the name of *prefix codes* related to *lossless data compression*. These two elements are both very important in cryptography concerning codes and data compression. In this particular case, we will see Huffman coding in the next section, a kind of lossless data compression in which it's possible to retrieve the information produced without losing any data.

The prefix codes that we will analyze now are important because they allow a word to be encoded uniquely, as in the following example:

character	a	b	c	d	e	f
code	0	101	100	111	1101	1100

Table 8.1: An example of prefix codes on letters

The encoding of the *abc* string, for example, will be *0·101·100*. We are sure in this way that if we are searching for a string starting with *abc*, we won't find any code other than *0101100*. This method is similar to a telephonic prefix index; we assume that the index we are calling is unique based on a country, region, or city. So, when we call a prefix number starting with *001*, for example, we are dealing with North America. By adding the next prefix, *415*, to *001*, we are calling a number in California, specifically the San Francisco area. In this way, telephone companies can divide the globe into prefix code numbers: *001* for America, *002* for Africa, and *003* for Europe, as you can see in the map in *Figure 8.12*.

The result is an easier way to search for a number located in a country-region-city code.

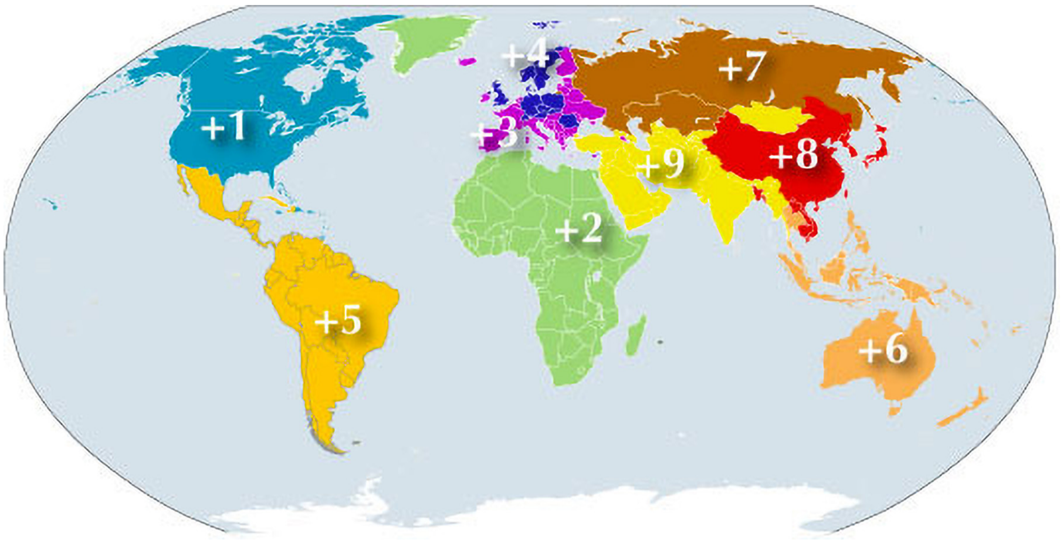


Figure 8.12: Prefix number codes across the globe

A search engine works similarly. Just like prefix numbers, the indexing algorithms have to be very efficient and accurate to search inside billions of instances, retrieving the information requested in a few milliseconds. Even if this book's scope is cryptographic algorithms, I want to go a little bit deeper to explain how tree graphs work because they are key to implementing an efficient search engine for encrypted data. So, let's go and implement a tree graph.

Huffman code

Suppose we want to solve the following problem: finding a way to represent text in binary code in the shortest way possible. It will be optimized, reducing the number of strings to as few as possible to encode text.

We have already seen a way to encode letters, numbers, and notations with ASCII code in *Chapter 1, Deep Dive into Cryptography*, in the *Binary numbers, ASCII code, and notations* section. But here, the problem is different: we want to be much quicker and more efficient in terms of encoding the text using as few bits as possible.

So, we can use an elegant method ideated by David Huffman, which operates by implementing a special tree graph.

There are 26 letters in the English alphabet, but not all the letters hold the same frequency in a text. We want to implement a tree graph that is able to codify, for example, six letters: *a*, *o*, *q*, *u*, *y*, and *z*: whose relative frequencies in a given text are as follows:

- *a = 20*
- *o = 28*
- *q = 4*
- *u = 17*
- *y = 12*
- *z = 7*

The objective is to discover the minimum possible number of bits, (*0*) and (*1*), to codify text made up of these letters.

Take a look at *Figure 8.13*, where you can find the entire strategy of this tree graph implementation:

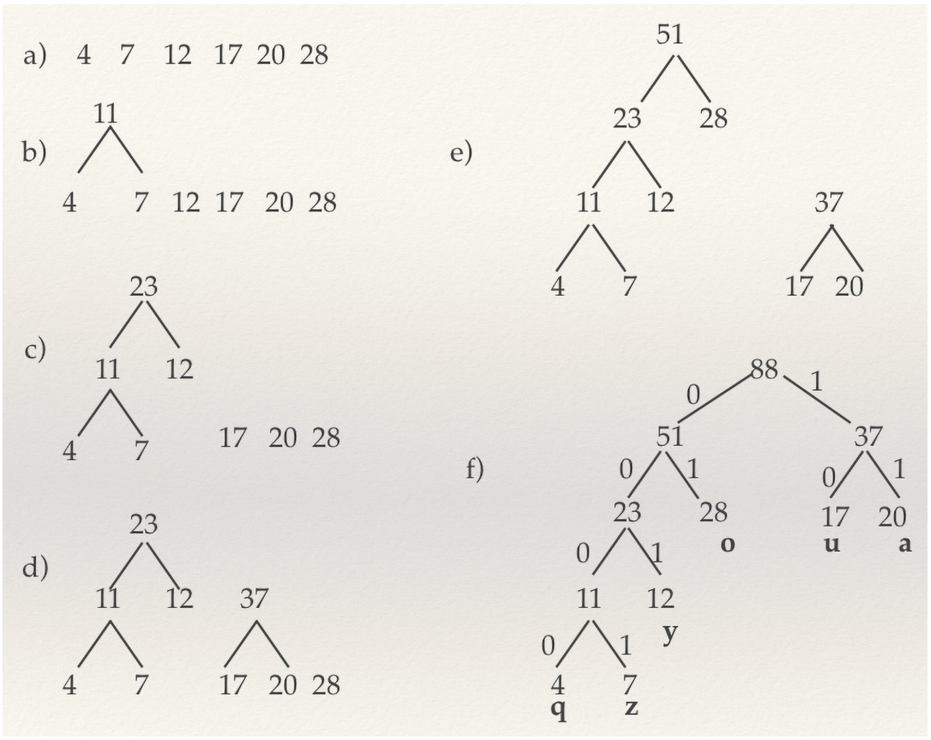


Figure 8.13: Huffman tree graph implementation

Now, we are going to analyze the sequence of steps necessary to implement the graph:

1. As the first thing, *a)*, we form a sequence of nodes, six in this case, disposing of them in ascending order based on the frequencies of the letters.
2. In stages *b)*, *c)*, *d)*, and *e)*, you can see the implementation of the main tree, where the nodes are partial trees of the same big tree, *f)*. They are composed of the sum of the number pairs, starting from the smallest, linked pair by pair until the top of the tree (see the final step, *f*). In order to have two nodes that link to 88, in step d, we create a new tree combining 17 and 20 into 37.
3. Finally, we find the summit of the tree represented by the number *88* (root), which is the sum of the vertices *51* (composed of 23 + 28) and *37* (17 + 20). Now that we have completed the tree, we can assign a value of *0* to all the left edges and *1* to the right edges (it also works in the opposite mode). Moreover, we assign to each letter a corresponding number related to its frequency: $q = 4$; $z = 7$; $y = 12$; $o = 28$; $u = 17$; $a = 20$.
4. After the tree has been built, we can read the *string code* for each letter (that is, for the letter *z*) starting from the root and add a *0* every time we go left to a child, and a *1* every time we go to the right. So, for example, to read the letter *z*, we start from the summit root *88* and count down how many instances of *0* are on the left and how many instances of *1* are on the right-hand edges we encounter before reaching the letter *z*, which is *0001*, in the following way:

$$q = 0000, z = 0001, y = 001, o = 01, u = 10, a = 11$$

Important note

It is possible to use other schemes to encode the tree. For example, we can assume that **0** will be on the right and **1** on the left.

The structure of the tree guarantees that any code will not be the prefix of another one. For example, $a = 11$ will be unique for any other letter encoded; no other letter will start with 11. Therefore, it is very easy for computers to read it and hasten the decoding process.

We can also verify that the total number of bits required to encode the six letters from the previous example, in this case, will be as small as possible:

$$(4 * 4) + (4 * 7) + (3 * 12) + (2 * 17) + (2 * 20) + (2 * 28) = 210$$

To encode the letter (*q*), it needs 4 bits, *q = 0000*, and its frequency is *q = 4*; the same bits for *z = 0001*, multiplied by *z = 7* (its frequency), and so on. We will make sure that the text is compressed as much as possible. For more in-depth insight, visit the web page on Huffman compression at `https://www2.cs.sfu.ca/CourseCentral/365/li/squeeze/Huffman.html`.

This prefix and compression code can help us significantly to compress the text and search on it, but these are not the only elements we must consider when implementing an efficient encrypted search engine.

Hash and Boolean logic

Another key element that needs to be considered is the *hash functions* that have to be combined with tree graphs. We have already covered hash functions in *Chapter 4*, *Hash Functions and Digital Signatures*, of this book. In the same chapter, we also saw more Boolean operators useful for hash functions in the *Logic and notations to implement hash functions* section. However, while in cryptography, hashes are employed for a digital signature to avoid exposing the content of the message, *[M]*, here, we use hashes for a different scope. The *object of a query* is also called the *keyword*. Another definition of a keyword could be the result of a process of indexing a string or a group of strings in a database.

If our search engine is efficient, it will index (a similar process to encoding) as few strings as possible, recognizing and discarding the double-indexed ones. In a search engine that works with encrypted data, the query itself is encrypted too.

In CSE, it's possible to search for complex queries made by multiple keywords. So, it's necessary to use Boolean logic operators such as *OR*, *AND*, and *NOT*, which we have already seen in *Chapter 2*, *Symmetric Encryption Algorithms*, in the section entitled *Notation and operations in Boolean logic*.

For example, if we want our search engine to retrieve the part of this book that contains this section, we will perform the following query:

[Hash and Boolean logic]

Once the whole content of the book has been encrypted and divided into chapters (we can suppose that each chapter is a file that is encrypted), we expect to receive an answer with the file (or files) that contains the three keywords (*Hash*, *Boolean*, and *Logic*), linked in this case by the *AND* Boolean operator:

[Hash "AND" Boolean "AND" logic]

In other words, it means to retrieve all the files (chapters) that currently contain the *Hash* + *Boolean* + *logic* keywords together (excluding, for example, the chapters containing either of the words *Hash*, *Boolean*, or *Logic*).

In this case, the proposition made by multiple keywords linked together by the *AND* Boolean operator will return this chapter, specifically this section, in an encrypted way. Now, with our private decryption key, we can decrypt and read the section's content.

That is how CSE works. Moreover, in our search engine, it's possible to set up search operations among encrypted data using all three operators, *AND*, *OR*, and *NOT*.

So, we can search, for example, in encrypted content, for a query such as the following:

[Boolean "AND "Logic "NOT" Hash]

This means that the search engine will search for all instances containing the words *Boolean* + *Logic* together, excluding *Hash*. In this case, we will be referring to *Chapter 2*, *Symmetric Encryption Algorithms*, where we find the *Boolean logic* section.

Queries in CSE are case insensitive; for example, if we type *hash* or *Hash*, the query results will be the same. Finally, you should have noticed that I have bracketed the query because even queries themselves in CSE are encrypted for privacy and security reasons.

The secret to searching for a keyword efficiently is to combine the index as a tree with hash functions and Boolean logic to obtain the most efficient path to the *target keyword*.

Our research team at Cryptolab has invented a unique technique for finding an *encrypted keyword* in the content of unstructured encrypted files in an average of 0.35 seconds semi-independently (-%) according to the number of bits processed.

Next, it's time to show you how CSE was created and what its principal functions are.

CSE explained

Data is often communicated across networks and stored on remote servers, which may be unsecured and non-private; eventually, it needs to be browsed, searched for, and manipulated regardless of the location where it is held. In the case of sensitive data (for example, in healthcare), it needs to be kept secure and private throughout the process. State-of-the-art technology regarding sensitive data management on remote servers achieves this objective via sub-optimal combinations: sensitive data is usually made secure by local encryption and then communicated and remotely stored.

In the event of requests to browse or search, it is decrypted on the remote server and then accessed. If manipulation is requested, additional encryption may even be necessary. This combination is functional but sub-optimal as it wastes computational power and exposes sensitive data in a clear-to-read form on remote servers (which are often provided by third-party cloud services). This problem could be prevented by employing the back transmission of encrypted data to local storage, where data would be safely decrypted, browsed, searched for, eventually encrypted again, and forward-transmitted to remote servers. It is clear how this solution protects sensitive data, but it results in longer processes, wastes bandwidth, and exposes data to multiple communications over unsecured networks.

In practical terms, this results in two very common behaviors: on the one hand, some private users renounce their privacy to take advantage of services and cost savings (this is typically the case with cloud services related to smartphones, which are appealing and inexpensive as long as the user is willing to share their private data). On the other hand, some institutions need to manage sensitive data, and since they cannot compromise in terms of privacy, they maintain privately managed data centers. When resilience to disasters is a requirement (typical for critical infrastructures in general), such data centers must additionally offer geographical and technological redundancy, thereby increasing costs even further.

This introduces the idea of *encrypting the data before outsourcing* to the servers and *always keeping the data encrypted*; thus, a potential hacker who overcomes all the security barriers and steals data cannot read it. The technical problem with encrypted data is that there are black boxes, and it's not possible to apply operations on encrypted data without decrypting it first. For this reason, it's a common practice to put this kind of data in a secure environment (such as a private data center) where it's possible to decrypt it without exposing it to hackers who can steal the data. This scenario is also not completely secure because a *trusted* person can access data, read it clearly, and copy and paste it to another media, such as a USB key.

In *Figure 8.14*, you can see a scheme of encrypted data in external storage. As you may notice, the encryption keys (the data key and the master key) are encrypted and stored in the external server (for example, the cloud) together with the encrypted data storage. The keys are used to unencrypt the data in encrypted storage when a query is requested. This process is necessary during normal encryption to perform searching in the database.

The shortcoming with this system is demonstrated when the data is decrypted to allow searching, thereby exposing the content.

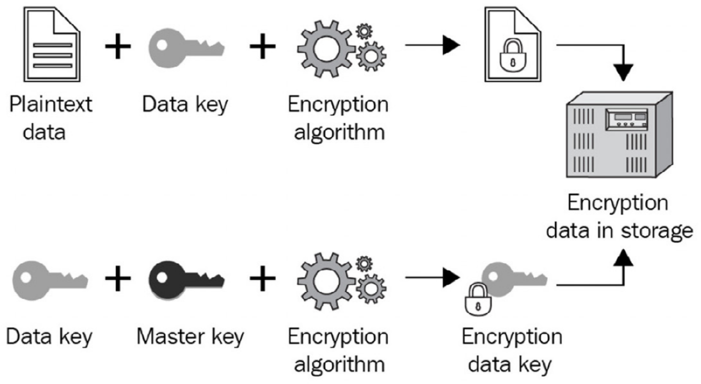


Figure 8.14: Encrypted data storage in the traditional way (not efficient and not secure)

Replacing this method with a method of searching blind makes the system wholly efficient and secure. That is what CSE can do, so let's see how it works and what its features are.

The innovation in CSE

CSE represents a technological solution based on isomorphism, *a transformation that preserves information*, as it can offer searching, browsing, and manipulation on encrypted data, that is, with *zero knowledge*; encrypted, sensitive data stored on public cloud providers will eventually be fully accessible by data owners only, still allowing cloud providers to offer searching, browsing, and manipulation over encrypted data by means of *encrypted queries*, thereby preserving zero knowledge. CSE is able to operate independently according to the kind of encryption that the user decides to adopt. On the one hand, private users could take back legitimate ownership of the value of their data and eventually trade it back to service providers through a fair deal, which is very different from the *take-it-or-leave-it* approach that we currently witness; this is a key value for the next generation of digital citizens who could also exchange or sell their data autonomously. On the other hand, public administrations, institutions, and critical infrastructures would be permitted to use a multiplicity of public cloud providers for safe and redundant data storage, with an important impact in terms of reduced costs and enabling access to tools for big data analysis.

The searching, browsing, and manipulation of encrypted data is a unique and unprecedented feature that will disrupt any business model where sensitive digital data is generated, communicated, stored, and processed.

CSE might constitute the technological architecture for *next-generation* search engines; data would be publicly available and searchable while preserving full encryption and integrity. New search engines would be perfectly aligned with EU/US legislation on data privacy (GDPR, which stands for General Data Protection Regulation) and recent socio-economic trends concerning data integrity, privacy, and protection. CSE scale-up will be vital for further deployment of the enabling infrastructure.

But let's now see how CSE works. In the scheme of *Figure 8.15*, you can find a good representation of the cycle accomplished by a query performed in an encrypted way that reaches encrypted content hosted, for example, by a public cloud provider:

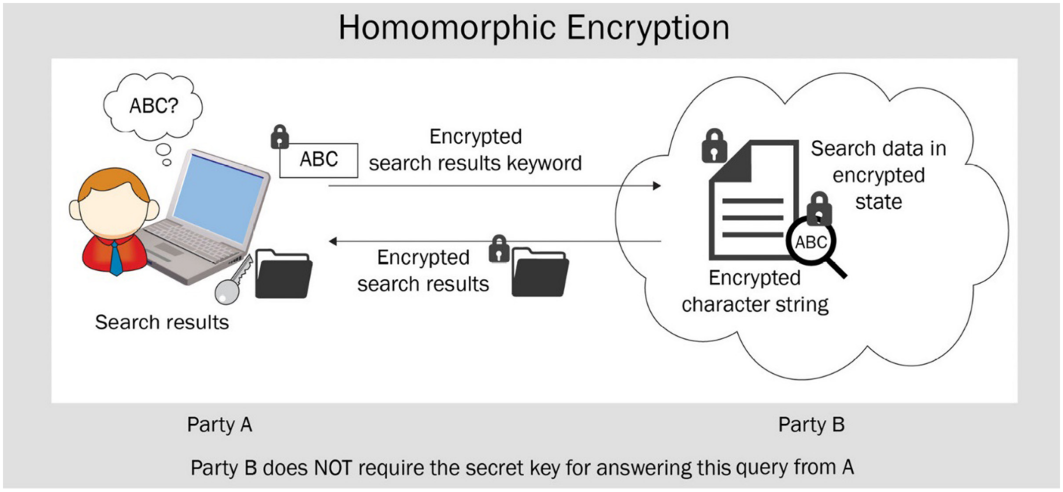

Homomorphic Encryption
ABC?
ABC
Encrypted search results keyword
Search data in encrypted state
ABC
Encrypted search results
Encrypted character string
Search results
Party A
Party B
Party B does NOT require the secret key for answering this query from A

Figure 8.15: The CSE scheme

In the scheme of *Figure 8.15*, there are two parties, but in reality, there is no exchange of information between the two subjects. Party A (here represented by a human) is generally a private server that hosts the encrypted keys and performs the query. The server asks party B (the cloud) to blind search for a query inside the encrypted content.

In *Figure 8.16*, you can see a simple scheme of a hybrid cloud architecture applicable to CSE:

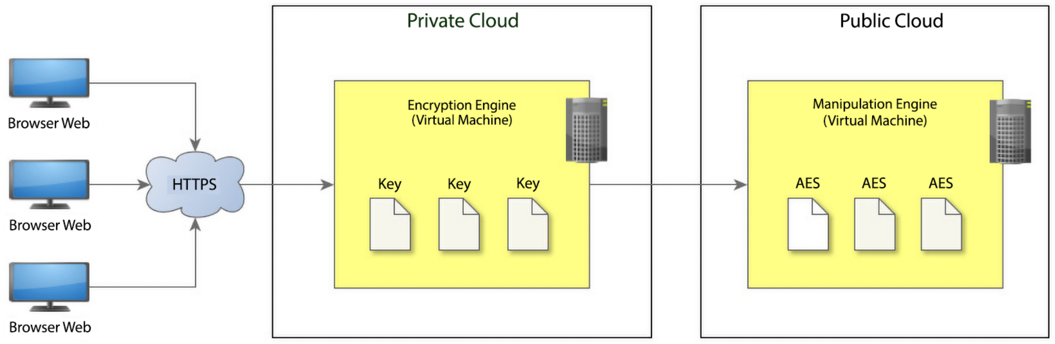


Figure 8.16: Basic hybrid architecture of CSE

In this basic scheme, the two virtual machines (VMs), **Encryption Engine** [EE] (which is the private server) and **Manipulation Engine** (**ME**) (which is located in the public cloud), are connected to a symmetrical encryption algorithm. In CSE, the system is agnostic to the algorithm used to encrypt the files. In the first version of CSE, AES-256 has been adopted (explained in *Chapter 2, Symmetric Encryption Algorithms*), but in the next version, a quantum cryptography algorithm will be adopted (besides classical cryptography) to send files through a quantum channel between the [EE] and (ME).

Important note

I've used the brackets to indicate the type of environment used in the VMs:

- [EE] stands for private protected environment (such as a private cloud).
- (ME) stands for public cloud or a non-protected environment.

Both the VMs have encrypted data stored inside them.

Generally, HTTPS (as you can see in *Figure 8.16*) is used only inside the protected private space to connect the user's devices to the [EE] located in a private cloud (that is, the company server, for example).

The [EE]'s primary function is to generate random encryption keys, all different, one by one, for each file encrypted. Hence, it will be impossible (-%) to find two files with the same encryption even if their plaintexts are identical.

The (ME)'s duty is to store the encrypted files and search through the encrypted content. In an efficient architecture, a search engine can be hosted in a third VM entirely separated by the [EE] and (ME).

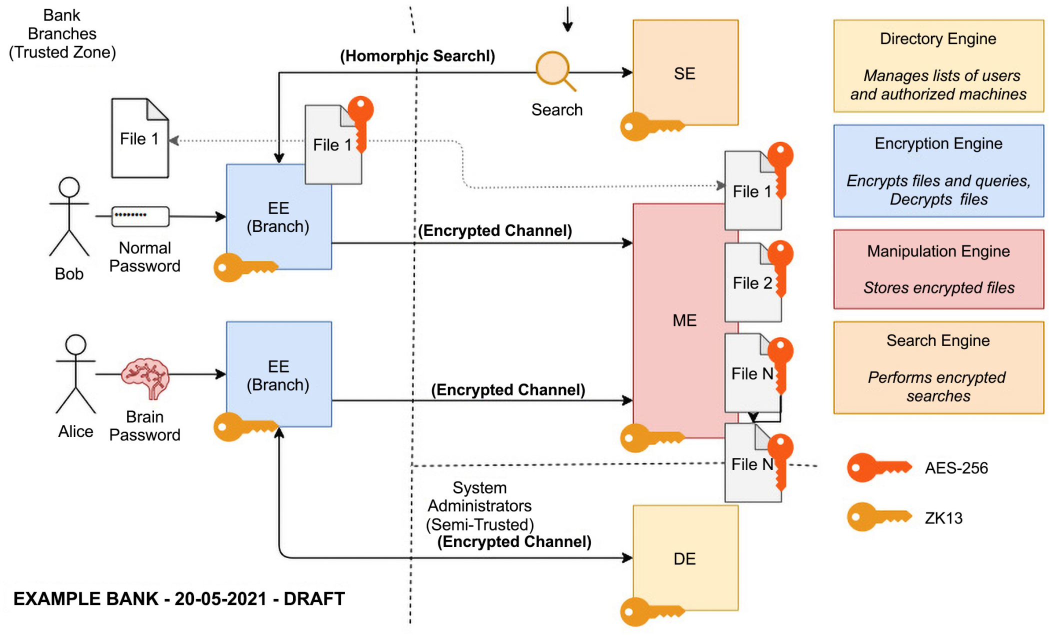


Figure 8.17: A complex architecture scheme of multiple [EE]s linked to the (SE), (ME), and [DE] in CSE

In the preceding *Figure 8.17*, the CSE architecture comprises several [EE]s connected to several (ME)s identified through a zero-knowledge protocol, which can provide *proof of identity* to the VMs' network. In the first version of CSE, a ZK13 non-interactive protocol was used (analyzed in *Chapter 5*, *Zero-Knowledge Protocols*), which controls the identity of the VMs, thereby avoiding possible man-in-the-middle attacks. In a future version, a special quantum protocol of authentication will be used to identify the VM in CSE.

Now that we have seen a working schematic of CSE, we will discover its computational grade of efficiency next.

Computational analysis on CSE

The secret to searching blind is to combine elements such as hash functions, encryption algorithms, authentication algorithms, and user interfaces into one helpful platform.

Computationally, the problem is to combine the best grade of security, given by the algorithms of encryption, decryption, and authentication, as well as the policies to log in and manage the platform, with the efficiency of searching in encrypted content. Suppose you want to search blindly inside the entire content of Wikipedia, encrypted, for example. In that case, the problem is the amount of content (expressed in kilobytes, gigabytes, or terabytes) and the number of *keywords* processed. In other words, we have to face off with a complex system given by the sum of the elements combined in CSE:

- Number of files encrypted
- Time to encrypt the files
- Bits per file and bits per total content processed
- Number of keywords processed
- Elapsed time per query
- Human-to-VM authentication
- VM-to-VM authentication
- Time to decrypt the file searched
- Time to wipe out the content from the memory of the system

All these elements have to be combined in an optimal solution given by an equation that I have processed to evaluate the *efficiency* (E) of CSE:

$$\boldsymbol{E = S/T}$$

Here, we have the following:

- E: Efficiency of the system
- S: Security of the cryptographic algorithms
- T: Time processed

The equation says that the higher the system's security level, and the less time taken to process a query, the more efficient the system will be.

In the equation proposed, 0 is the minimum security level of an algorithm and 1 indicates 100% security. Let's suppose now that the system doesn't adopt any encryption to protect the data, so then we have *security (S) = 0*, as opposed to if we use quantum cryptography encryption, where we would probably have a very high security level, *S = 1*. The searching time, *T*, is a variable that goes from 0+ to infinity, even if we know that an acceptable searching time answer is at most two seconds per query. Hence, the maximum degree of efficiency will be very high security, for example, *S* = 0.999999, and the lowest time per query as possible, *T = 0.00001*. So, if we want to attain the highest efficiency, we should augment security and reduce time.

Note

Here, we suppose that data is transmitted and processed through the internet, so it can be exposed to external attacks and be spied upon. However, it's not said that a system is more protected because data is encrypted. If a system works offline and information is not shared or processed externally from the offline computer, the system could have a higher degree of security even if its content is not encrypted.

The (0+) notation stands for the lower limit of *T* that tends to zero but cannot be zero.

Since time is a deterministic and estimated variable, the problem is how to estimate the best grade of security and degree of efficiency.

In CSE, AES-256 is adopted among the other algorithms as a symmetrical encryption algorithm to transmit the files between the [EE] and (ME). AES, as we saw in *Chapter 2*, *Symmetric Encryption Algorithms*, is now a standard in symmetric encryption (valid at this time, although the same cannot be said for the future).

I have tried to calculate the grade of the complexity of AES-256 bits in terms of attack and computation, related to the power of calculation at our disposal. Obviously, things are evolving fast, and what is considered secure now may not be in three or five years. Here, we are only looking at computational power, but from a security perspective, you could follow options such as encrypting with Twofish.

Let's try to follow the logical reasoning in the next section about how to calculate the efficiency of a cryptographic algorithm such as AES-256.

Example of computational brute-force cracking

Let's suppose we have a four-core machine, let's say a MacBook Pro 2015 i7 core. This machine equates to 1,024 MiB per second or 2^{30} bytes per second.

As AES-256 uses a 16-byte block size (2^4), on average, a single high-performance PC can perform a calculation of $2^{(30-4)} = 2^{26}$ blocks per second. Supposing that only one computer is always running for *60 sec * 60 min * 24 h * 365.25 days = 31,557,600* (seconds in 1 year), the total computation per year related to a single high-performance computer will be $31{,}557{,}600 * 2^{26} = 2{,}117{,}794{,}686{,}566{,}400$.

That is the number of keys we have to explore, approximately 2,117.8 trillion.

To perform a brute-force attack on AES-256, on average, you will need to try 2^{255} keys.

Important note

You could be lucky and try fewer than 2^{255} keys, but it remains an incommensurable number.

So, we have to divide 2^{255} by *2,117.8 trillion*. The result is a number expressed as an exponent of 10, which is about $2.73 * 10^{61}$. Written in full, this is *27,337,893,038,406,611,194,430,009,974,922,940,323,611,067,429,756,962,487,493,203* years.

This number can also be represented as 27 trillion trillion trillion trillion trillion years. Just to give you a comparison, the Universe has only existed for 15 billion years!

What would happen if all the computers on Earth were to join together to crack AES-256?

If you are passionate about computation or other strange questions about big numbers, you can try to use the *Wolfram Alpha* search engine and ask, "*How many computers exist on Earth?*"

Alpha is a high-performance search engine supported by Wolfram Mathematica as a computational platform. You can check, and the answer you will receive is 2 billion PCs on Earth.

You can also check the entire process of my previous argumentation using *Wolfram Alpha*. Moving on, we divide the time employed by 1 computer by 2 billion, assuming that all the computers on Earth have the same computation of a four-core MacBook Pro i7. Obviously not, but I am optimistic.

The result will be *13,650,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 years*.

Or, expressed in a power of 10, it will be *$1.365*10^{52}$* years. This is an incommensurable computational time, even for all the computers on Earth.

Based on this thesis, we should assign the efficiency equation S = 1 to AES-256. But things are not that simple, and I will demonstrate why with a simple question:

What if, instead of combining all the power of classical computers, we use a quantum computer?

As we will see in *Chapter 9*, *Quantum Cryptography*, Shor's algorithm can crack the factorization problem. However, AES doesn't rely on this mathematical problem. So, isn't a quantum computer theoretically able to crack AES?

As you can probably imagine, cracking AES is a problem related to searching for something in a vast amount of content: the amount of 2^{255} keys hidden inside the algorithm. **Grover's algorithm** could be a good candidate (if it can be implemented on a quantum computer in the near future). Indeed, all algorithms, such as AES, that rely on hidden keys, basing their strength on the confusion and diffusion principle of Shannon, can theoretically be broken by a quantum computer.

I will talk about Grover and quantum search in *Chapter 10*, *Quantum Search Algorithms and Quantum Computing*. Probably, in the future, a better algorithm will come out for Quantum Search. If that happens, it will not be only a matter of adding or doubling the key size but something that our classical logic and mathematics will not be able to handle.

Important note

Grover's algorithm shows how to search in unstructured content with a quantum computer.

The good thing is that CSE is a system projected to be agnostic to any algorithm used. Theoretically, it can survive quantum attacks, replacing some algorithms inside its core system, even the *homomorphic search algorithm invented by Cryptolab's team, HK16,* to perform queries on the encrypted content. That is, the core of the system will be replaced. It can be substituted by *quantum homomorphic searching*, which can work on a quantum computer more efficiently.

Finally, I wish to answer the question: what is the current efficiency of CSE?

It is not up to me to give CSE a score because I am one of the system's designers. But I can certainly say that given the algorithms adopted and the time elapsed per query, with an average time of 0.35 seconds, CSE has superior efficiency both in terms of current security and time. That will remain valid until the conditions change, as we have seen.

Now that we have analyzed the efficiency of a system such as CSE, we are going to see the main applications of such a system in real life.

Applications of CSE

Most applications of CSE fall among **encrypted data storage** and **file synchronization and sharing (FSS)** in the cloud. As I have said, encrypted data storage in the cloud holds an important added value compared with other cloud applications: **homomorphic search**. An interesting use case of CSE applied to blockchain is **blind payments**. Here, the holomorphic search engine has the important function of listing the system's client anonymously and securely processing transactions. In the healthcare field, two applications adopting CSE have already been tested: the management of **electronic health records (EHRs)** in the cloud and **CovidFree**. Going a little bit deeper, CSE technology enables data search and big data analysis in EHR systems while preserving patients' security and privacy. CovidFree is an application patented in 2020 to fight the COVID-19 pandemic that maintains the privacy and security of data to monitor the virus. This application monitors and virtually contains the virus through the control of access in public places (airports, trains, buses, restaurants, and so on) without using any sensitive data to track people.

A scheme for sharing data in an encrypted environment is represented by the following diagram:

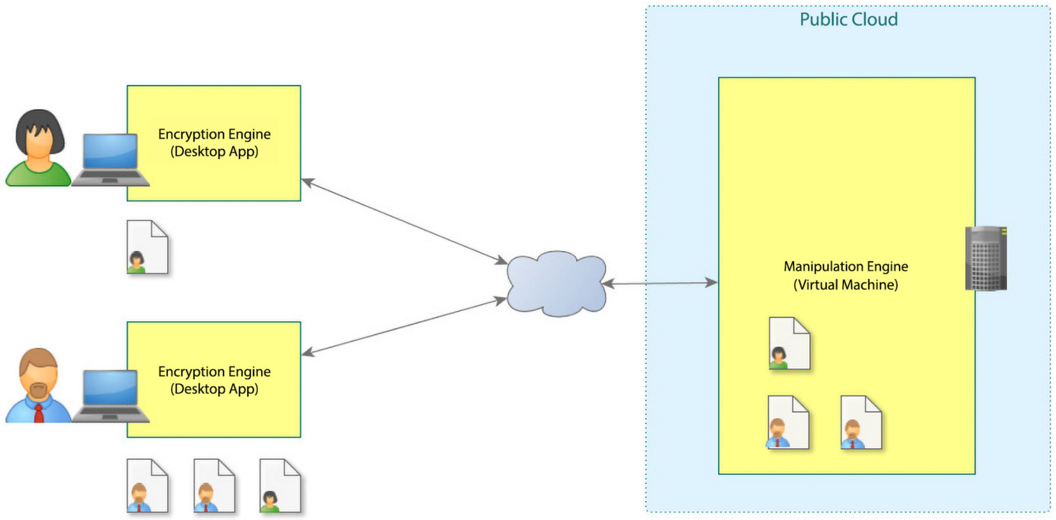

Public Cloud
Encryption Engine (Desktop App)
Encryption Engine (Desktop App)
Manipulation Engine (Virtual Machine)

Figure 8.18: Sharing data in the encrypted cloud

Other applications of CSE are related to video surveillance and video image recognition through AI. CSE technology, in this case, enables video management and real-time analysis of the encrypted data directly, thereby preserving the privacy of sensitive data within the video domain.

As you can see in the following diagram, CSE applies to different fields in IT:

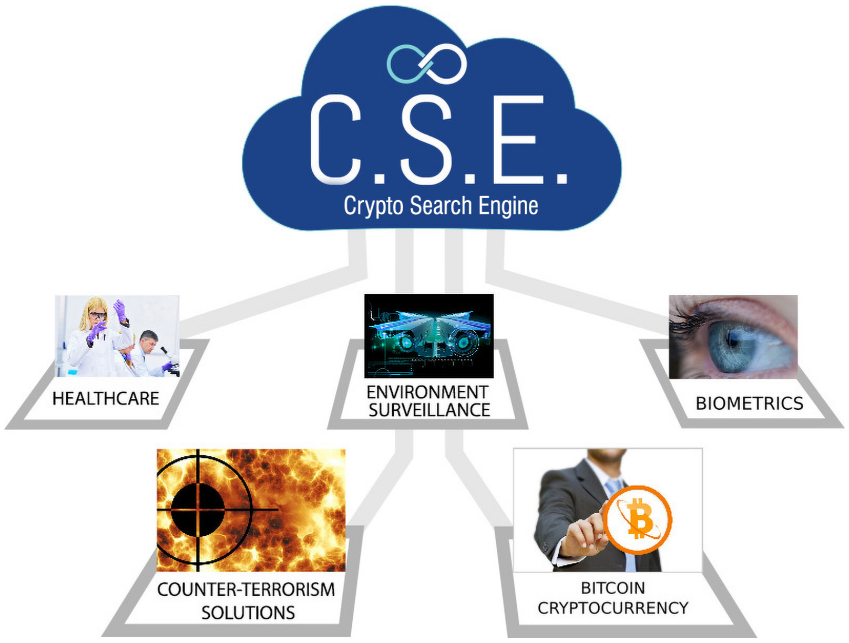


Figure 8.19: Some fields of application of CSE

Another interesting use case is in the automotive field. In the future, more and more connected cars and autonomous cars will transmit data among themselves to understand, for example, when running on a particular route, whether an accident or a traffic jam will occur. In this scenario, it is easy for hackers to take control of the car if the data transmitted is in plaintext. CSE prevents this (or at least reduces the chances of it) because data is always transmitted and processed in an encrypted way. Even if hackers are sniffing around the communication channel, they won't find any readable messages; they'll just find strings of encrypted messages.

In biosciences, a scenario where CSE can be very useful is DNA storage and sequencing. This is a very sensitive field that combines the possibility of discovering sickness and many characteristics of a person, so it has to be managed very carefully. Like all data related to our wellness, physical, and psychological information, CSE can store the data collected in the DNA of a person and search for it blind, preventing it from being stolen or the information from being spied upon by an unauthorized party.

Figure 8.20: DNA storage can be processed blind using CSE

Most of the applications described here have already been tested and some are in use in real life. The hope is that the next generation of search engines will guarantee privacy and security to everyone, and CSE could be an example.

The new frontier of CSE and a new quantum algorithm for message transmission: QTM

In the future, we aim to adopt a new quantum algorithm I have called **Quantum Transmission Message (QTM)** to perform symmetric encryption and data transmission for CSE. This algorithm is now under testing and will soon be available in a beta version.

We will be substituting AES with quantum data transmission of photons based on a new, revolutionary concept: it is possible to transmit not only the key with a quantum algorithm (BB84, for example, as we'll see in *Chapter 9*) but it is possible to also transmit the message encrypted with a quantum system of my invention: QTM.

This algorithm, invented and patented by me in 2023, will soon be released for government, ground communications, and satellite telecommunications. It will be valid not only for CSE encryption/decryption but will be available for any QTM performed both on Earth and on satellite communications.

Hopefully, after the tests we will do in our quantum computing laboratory, I will report the results of this new quantum transmission algorithm to the community. Therefore, I invite you to book a virtual visit to our new laboratory of quantum computing and quantum cryptography, unlocking the code you find inside this book.

Summary

In this chapter, we introduced the basic principles of homomorphic encryption and its fundamental bases, and we analyzed partial homomorphism in RSA.

After that, we explored the math behind search engines and, in particular, Eulerian circuits. We then undertook a deep dive into tree graph theory, experimenting with a particular tree graph construction such as Huffman code. If usefully combined with hash trees and other elements, this algorithm is an excellent candidate for searching and compressing data in unstructured content.

Finally, we analyzed CSE, looking at its principal elements and possible future applications.

You have now learned the fundamentals of CSE, which can be a springboard for learning more. If you go on to research how it is implemented, keep in mind that searches are not performed in clear text, but on storage. The storage is encrypted with AES and needs to be manipulated to be searched. To properly learn about that, however, is a task for another day.

Now that you have learned about CSE and have explored most of the main cryptographic classic algorithms, we will approach a new dimension that is opening up in cryptography: quantum mechanics and quantum cryptography. These will be explained in the next two chapters.

Section 4

Quantum Cryptography

This section finishes the book with one of the biggest topics in cryptography, quantum computing. Quantum computing's ability to perform brute force attacks leaves many security measures vulnerable, so it is an essential topic for any future cryptographer. In order to understand how elements of quantum computing are applied to cryptography, we will look at the Grover algorithm as an example of quantum search.

This section of the book comprises the following chapters:

- *Chapter 9, Quantum Cryptography*
- *Chapter 10, Quantum Search Algorithms and Quantum Computing*

9

Quantum Cryptography

In this chapter, I will explain the basics of **quantum mechanics** (**Q-Mechanics**) and **quantum cryptography** (**Q-Cryptography**). These topics require some knowledge of physics, modular mathematics, cryptography, and logic.

Q-Mechanics deals with phenomena that are outside the range of an ordinary human's experience. So, it might be difficult for most people to understand and believe this theory. We will start by introducing the bizarre world of Q-Mechanics, and then examining Q-Cryptography, which is a direct consequence of this theory.

We will also analyze quantum computing and Shor's algorithm. Shor's algorithm is quite complex, so I have divided it into five steps: initializing the qubits, choosing a random number, performing the quantum measurement, finding the right candidate, and then factorizing. We will, of course, go through it step by step.

Finally, we will look at which algorithms are candidates for post-Q-Cryptography.

So, in this chapter, we will cover the following topics:

- Introduction to Q-Mechanics and Q-Cryptography
- An imaginary experiment to understand the elements of Q-Mechanics
- An experiment on quantum money and **quantum key distribution** (**QKD**) (BB84)
- Quantum computing and Shor's algorithm
- The future of cryptography after the advent of quantum computers

Let's dive deep inside this fantastic world, based on strange and (sometimes) incomprehensible laws.

Introduction to Q-Mechanics and Q-Cryptography

So far in this book, the kind of cryptography we have analyzed always followed the laws of logic and rigorous mathematical canons. Now, we are approaching it with a different logic. We are leaving the logic of classical mathematics and landing in a new dimension where the classical physical events, as we know, are completely different and, in many cases, counterintuitive.

We have to face off with a kind of science that Niels Bohr (one of the fathers of Q-Mechanics) said this about: *"Anyone who can contemplate quantum mechanics without getting dizzy hasn't understood it."* Einstein's divergence defined the entanglement theory as a *phantasmatic theory*.

A couple of preliminary considerations about Q-Cryptography and Q-Mechanics are as follows:

- All the cryptography you have learned about until now, even the most evolute, robust, and sophisticated, relies on two elements:
    a. The difficulty of breaking it depends on the algorithm and its underlying mathematical problem: factorization, discrete logarithm, polynomial multiplication, and so on.
    b. The size of the adopted key. In other words, the length of the key or the field of numbers that the algorithm operates in to generate the key defines the grade of computational break-even under which any algorithm fails. For example, if we define a public key (N) in RSA as $N = 21$, even a child in elementary school will be able to find the two secret numbers (private keys), $p = 3$ and $q = 7$, given by the factorization of the number; that is, $21 = 3*7$. Conversely, if we define a field of operations of 10^{1000}, then the factorization problem becomes a hard problem to solve.
- In Q-Cryptography, the problem doesn't rely on any of these elements. There isn't any *mathematically hard problem to solve* that underlies this kind of cryptography (in the classical sense) but everything is based on a particular, and even bizarre, hypothesis: in Q-Mechanics, an element (a particle, such as a photon) can be in a state of *1* and *0* at the same time. It can be present in two places at the same time. It is impossible to determine the state of a particle until it's measured. As we'll learn shortly, the concept of *time* itself in Q-Mechanics loses its meaning because the causality property is not an option, just a mere possibility.

Before we analyze the hypothesis that was anticipated in this preamble, let's start with an experiment that changed the way that people thought about microparticles.

An experiment that changed the story of quantum

The first experiment that changed the history of science in this branch forever was created by Thomas Young, a polyhedric scientist, at the end of the 18th century. Besides that, Young was the first person to decode some hieroglyphics, so we can also count Young among cryptographers. He often used to walk beside a small lake where groups of ducks swam. He noticed how the waves that were created by the ducks while they swam interacted with themselves and intersected with each other, generating ripples. This reminded him of the same behaviors of waves of light, which inspired his work.

The experiment, as shown in the following diagram, demonstrates that light waves behave like particles and cause multiple small stripes on the screen on the back of two slits where the light enters, instead of generating only two large strips.

So far, there seems to be nothing paradoxical about this experiment. However, if we run it by shooting just one photon—a nanoparticle of light—through the two slits at regular intervals (for example, one every second), the result is unexpected. Instead of producing two dark lines on the screen behind the slits, as one might anticipate, it creates a long row of striped lines. These lines resemble the patterns generated by light waves.

In the following diagram, you can see Young's experiment reproduced:

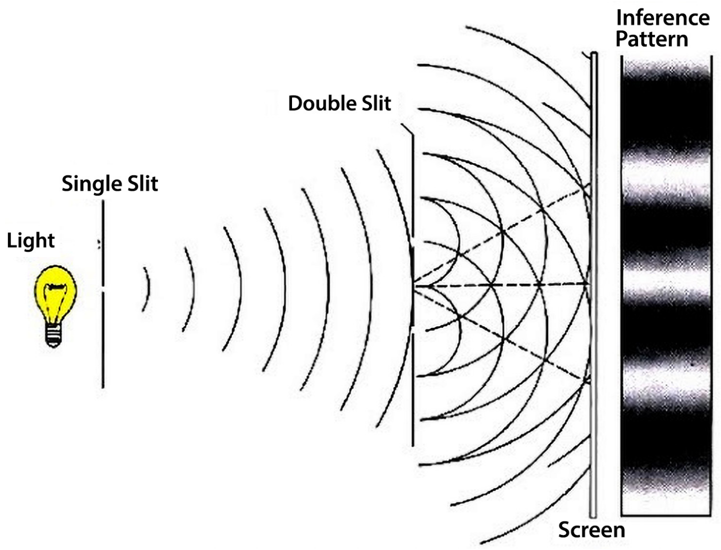


Figure 9.1: Young's light experiment

In the next section, I will provide a variant of this experiment that has been reinvented for you to understand Q-Mechanics (or more likely to get you dizzy).

An imaginary experiment to understand the elements of Q-Mechanics

In this section, we will try to go deeper by providing an imaginary experiment to introduce Q-Mechanics. This experiment's elements are similar to Young's experiment: a wall with two slits and a screen on the back. Throughout the chapter, step by step, we will address more elements that will help you set up the experiment and understand the paradoxes of Q-Mechanics.

Suppose we shoot a photon toward the two slits shown in the following diagram. We will think of the photons as small marbles.

Step 1: superposition

Let's start by shooting marbles that are a normal size. Each time a marble is shot, it will likely move toward the slit and pass through it. So, let's assume that at the end of the first round, the result will be that most of the marbles will hit the screen in a straight line behind the slits:

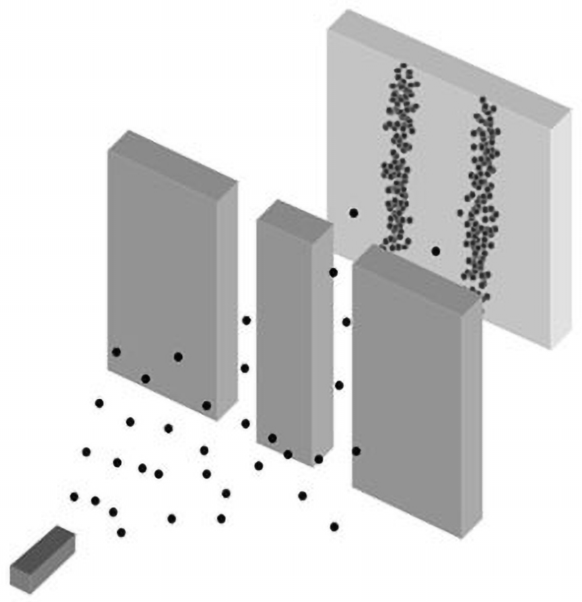


Figure 9.2: The marbles hitting the screen, causing two separate lines at the back

Now, suppose that the two slits become very narrow, and the marbles become very small. This is the case when we are dealing with photons, and we will see that the result is very different. Unlike the results with normal marbles, marbles with the same dimension as a photon produce a striped pattern. Let's take a look at the following diagram:

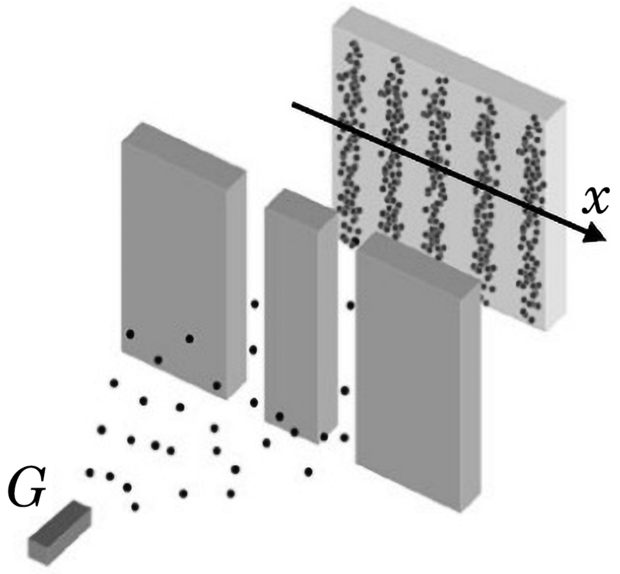


Figure 9.3: The marbles are very small and cause a striped pattern

As we can see, after the measurement, particles produce a striped pattern, as shown in *Figure 9.3*, provided that both the slits and the particles are small enough. The result of this phenomenon is known as **waves**.

When a wave passes through a slit, it spreads out across the other side. If there are two slits instead of one, then two waves are produced that interact with themselves. These waves will produce the effect of a striped pattern, as we can see on the screen that was reproduced in Young's experiment, shown in *Figure 9.1*.

The problem with this representation is that if we only launch one marble (particle) at a time, it still produces a striped pattern. (In short, the reason for this is related to the volume of energy related to the particles.) So, how is it possible for a single marble, however small it may be, to pass simultaneously through the two slits?

Let's stop just for a second to reflect on this first important property. If we give the state of *(0)* to the photon that passes through the left slit and *(1)* to the photon that passes through the right slit, then we can say that the two photons are passing simultaneously through the two slits, so they are in a state of *(0)* and *(1)* simultaneously. This is the first important property in Q-Mechanics and it is called *superposition*.

If we block one slit and shoot the marbles just through one slit, the pattern that's generated will be similar to a normal marble hitting the screen, just behind the open slit directly. But if we unlock both slits, the striped pattern returns.

Superposition can offer a vision in which a particle can be in different states at a determinate moment. However, another part of physics, and some scientists, have a second interpretation, called the **Copenhagen interpretation** or **many-world interpretation**, of this experiment. This interpretation lies not in superposition, but in the equally bizarre concept of the *multiverse*. In this concept, the different states that a particle is in are represented by many universes.

This alternative vision of a multiverse offers a philosophical theory of quantum reality that can even be transposed into our lives. There could be copies of yourself existing in a plentitude of states, playing different roles in different parallel worlds.

Anyway, given the scope of this book, we will continue to experiment and talk about another important characteristic of Q-Mechanics: indeterminacy.

Step 2: the indeterminacy principle

Now, let's test what happens if we put a detector in front of each slit. We should expect both the detectors to reveal the marbles simultaneously because, as we saw earlier, shooting only one particle makes it appear in a superposition state; however, this is not what happens here.

If we shoot a marble small enough to be compared to a small particle, we will see that the marble only passes through one of the two detectors, never both. Any attempt to discover which of the two slits the marble passes through generates *indecision*, which causes the end of the striped pattern. So, this attempt forces the marble to pass through one slit, not both.

That is another fundamental property of Q-Mechanics: when the photons are in a state of superposition (or are playing in a multiverse), they are *(0)* and *(1)* at the same time; instead, when they are observed, the photons change their behavior and choose to be *(0)* or *(1)*.

An **observer** in quantum mechanics is anything that detects a quantum particle. An observation is also called a measurement, since it is an act of collecting data.

You can understand that the indeterminacy property is strictly correlated to superposition. We will return to this later, but for now, let's continue with our experiment.

Important note

In this thought experiment, we will simplify things by pretending to see photons with our eyes. That is, of course, impossible; in the real world, we can only observe photons using detectors. In reality, the determination of a superposition state is made by a quantum measurement, which is the interaction between the quantum system and the measurement device.

If we close our eyes and don't observe the marbles passing through the slits, even if the detectors are put in front of the slits, the superposition state is still activated. Only when we open our eyes and look is the state of indeterminacy stopped and the marble is forced to take a position of *(0)* or *(1)*. This means that the marble we are dealing with should be different from waves.

The state of a classical wave is **deterministic** and produces its effects in any condition, regardless of whether we observe the experiment or not. In Q-Mechanics, things are different; particles act in strange ways guided by rules that are outside classical physics. This problem produces another interesting implication in physics, relating to philosophy, but that is beyond the scope of this book.

Now, suppose we put two objects behind the two slits. While shooting the marble through the slits, we can imagine that the marble hits one of the two objects. This is true when we look at the situation, but again, if we close our eyes and don't look, then the probability that the marble will knock down one or the other object is the same. In other words, the two objects can be compared to the famous thought experiment of *Schrödinger's cat*, where the cat is alive and dead at the same time until we discover it.

According to the mathematics that describes the probability of waves, neither outcome is certain. Only when we open our eyes and look can we know whether an object is alive, *(1)*, or dead, *(0)*. Moreover, the fact that we can only assign a certain *probability* to a particle situated in a certain place in the universe and that it comes out only when someone looks at it is logic that's very hard to understand and believe.

Note again that by "look" I am speaking in terms of human eyes because of our experiment, but in reality, this is a quantum measurement taken by a device.

However, despite its crazy logic, all the experiments we've looked at until now demonstrate that the indeterminacy property is valid! **Probability** is another fundamental element that we have to keep in mind when we deal with Q-Mechanics. The probability of objects being in one state or another and one place or another at the same time induced Einstein to be a strong adversary of the quantum theory and, in particular, of Schrödinger. Regarding this logic, Einstein commented, *"God does not play dice with the Universe."*

When we talk about probability in Q-Mechanics, a key property that exhibits a particle's probabilistic nature is spin. So, let's explore what spin is and why it is so important for our studies.

Step 3: spin and entanglement

To explain how a particle could be in two states simultaneously in Q-Mechanics, we need to introduce **spin**.

In literature, spin refers to *the intrinsic angular momentum of a body*. It can be combined with the orbital angular momentum to calculate the total angular momentum.

The direction of the spin of a particle can be described as an imaginary arrow crossing the particle in different directions. Particles spinning in opposite directions will have their arrows pointing in opposite directions, as shown in the following diagram:

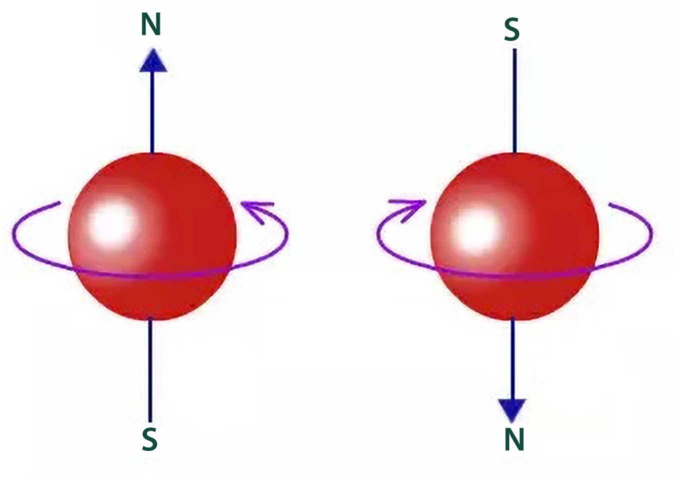


Figure 9.4: Spin

Spin is very important for Q-Cryptography, as you will understand later when we analyze how to quantum exchange a key in more detail. The content or information that a particle can carry depends on its spin. This depends on the information that's enclosed in the spin direction; that is, whether Alice and Bob can determine the bits of a quantum-distributed key. But we will look at this later.

Now, let's return to the concept of quantum information: we know that information in the classical world is made up of letters and numbers, but we can give or receive pieces of information in a discrete way, such as *YES* or *NO*, *UP* or *DOWN*, or *TRUE* or *FALSE*. We can also *discretize the information*, reducing it to a bit in the real world. We can do the same in Q-Mechanics, but we call it a **quantum bit (qubit***)*.

Let's look at an example of a coin with heads or tails, referring to a physical entity that could represent a bit in a state of *(0)*, meaning heads, or *(1)*, meaning tails. In qubits, we use the **Dirac** notation, which describes the state of a bit using "bra-ket" notation:

$$|0> \qquad |1>$$

We already know that in superposition, the state of a particle can be *1* and *0* simultaneously, but when observed, the particle is forced to find a position. This phenomenon is called the **collapse of the wave function**. This phenomenon occurs when the waves (initially in a superposition state) reduce to a single state. This is caused by an interaction with the external world; for example, someone who spies on the channel of communication between the two particles.

When a pair of particles interact by themselves, their spin states can get entangled, which is what scientists call **quantum entanglement**.

Broadly speaking, **quantum entanglement** occurs when a group of particles are no longer independent of the state of another group. This occurs when the groups interact with one another or share proximity.

When two electrons become entangled, the two electrons can have opposite spins, as we will see soon in the example proposed. This means that if one is measured to have an *up* spin, the second one immediately becomes a *down* spin:

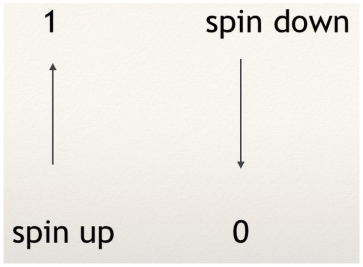


Figure 9.5: The spin "up" and "down" of an electron

Even if we separate the two electrons so that they're arbitrarily far away and measure the spin of one, we will immediately know the spin of the second one. For example, if we measure an electron's spin in a California laboratory and know it's *up*, then we will know that the other one in the second laboratory in New Jersey will be *down*. It doesn't matter how far apart the two particles are.

So, we can say that the determination of an entangled state occurs faster than the speed of light!

This theory had Einstein as a fierce opponent, who called this phenomenon "spooky action at a distance." But despite what Einstein said, entanglement is very useful for our technological world, as we will see very soon.

Now, let's explore an experiment that can demonstrate that the entanglement of information is real.

In 2015, a group of scientists, led by Ronald Hanson, from Delft University of Technology in the Netherlands, set up an experiment to demonstrate that two particles that are entangled could communicate between themselves faster than the speed of light.

The experiment was carried out by setting two diamonds in two different laboratories, *A* and *B*, separated by a distance of 1,280 m. An electromagnetic impulse was shot to cause the emission from each point, *A* and *B*, of a photon in an entanglement state with an electron spin. The experiment demonstrated that when the two particles arrived simultaneously at a third destination, *C*, where a detector was installed, their entanglement was transferred to the electrons. We can see the experiment location mapped out in the following figure:

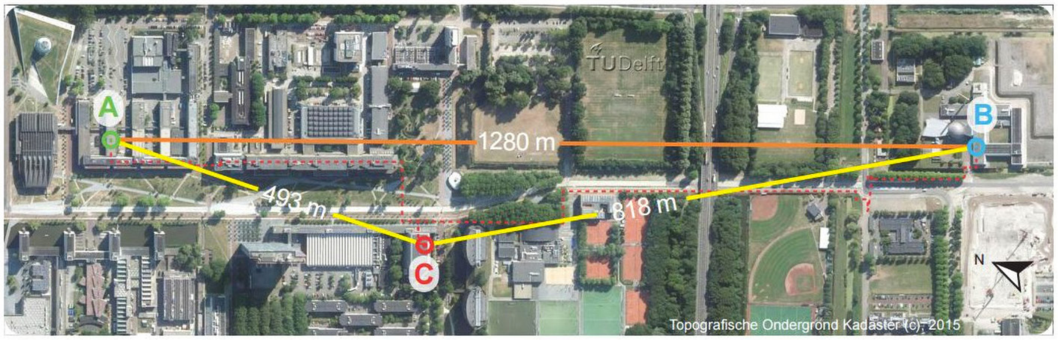


Figure 9.6: Entanglement experiment (Delft University of Technology)

Recently, the group achieved an important technical improvement: the experimental setup is now always ready for **entanglement on demand**. This means that the entanglement state between two particles can be obtained in the future on request, now allowing the development of quantum applications that were probably considered impossible earlier. If you are interested in learning more about this experiment, you can refer to the official Delft University website: `https://www.tudelft.nl/2018/tu-delft/delftse-wetenschappers-realiseren-als-eersten-on-demand-quantum-verstrengeling`.

Some examples of fields of action where this bizarre entanglement phenomenon is already used are as follows:

- Entangled clocks and all the applications behind the stock market and GPS
- An entangled microscope, built by Hokkaido University, to scrutinize microscopic elements
- Quantum teleportation, involving transporting information
- Quantum biology for DNA and other clinical experimentations
- Our object of study: Q-Cryptography

So, it's just a combination of all these elements and properties to create a disruptive and fascinating application in **Quantum Key Distribution** (**QKD**).

Before we look at this in more detail, let's talk about how this process came about and the elements that are combined in this process.

Origin of Q-Cryptography: quantum money

Now that you have learned about the fundamentals of Q-Mechanics, we will talk about Q-Cryptography. The curious story of its first application began in the 70s, when a Ph.D. candidate, Stephen Wiesner from Columbia University, had an idea. He invented a special kind of money that (theoretically) couldn't be counterfeited: **quantum money**. Wiesner's quantum money mostly relied on quantum physics regarding photons.

Suppose that we have a group of photons traveling all in the same direction on a predetermined axis. Moving in space, a photon has a direction of vibration known as the **polarization of the photon**. The following diagram shows what we are talking about:

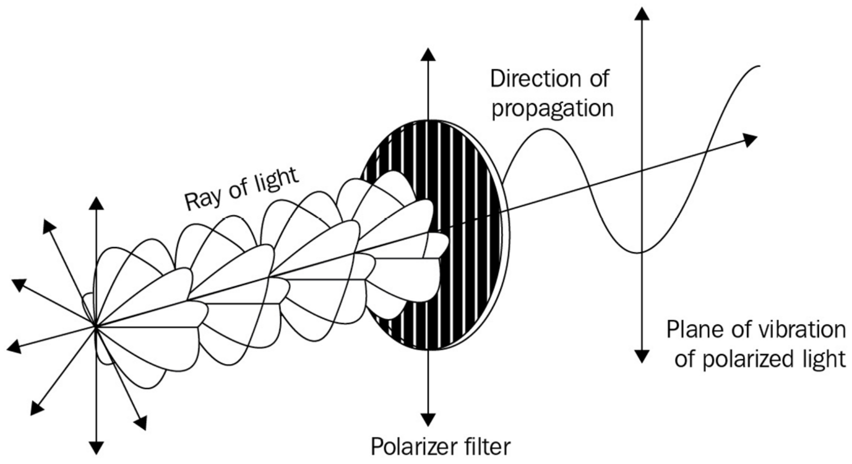


Figure 9.7 — Photon polarization

As you can see, photons spin their polarization in all directions, including diagonals, as the initial state of an unpolarized photon is the superposition of oscillating horizontally and vertically. However, if we place a filter, called a **polaroid**, oriented vertically, we will observe that photons oriented vertically pass through the filter 100% of the time.

In particular, the filter will block photons polarized perpendicularly to the filter 100% of the time, while photons oriented diagonally in relation to the filter will pass (randomly) about 50% of the time. Moreover, these photons have to face a quantum dilemma: passing through the filter, they have to be observed, so they have to *decide* on their direction – whether they want to assume a vertical orientation or a horizontal one.

This trait leads to some odd results when looking at an experiment with polarized lenses. As shown in the following diagram, it's weird that if you add one more lens to the two already there, you can see more light than before (see *Polarizer 3*):

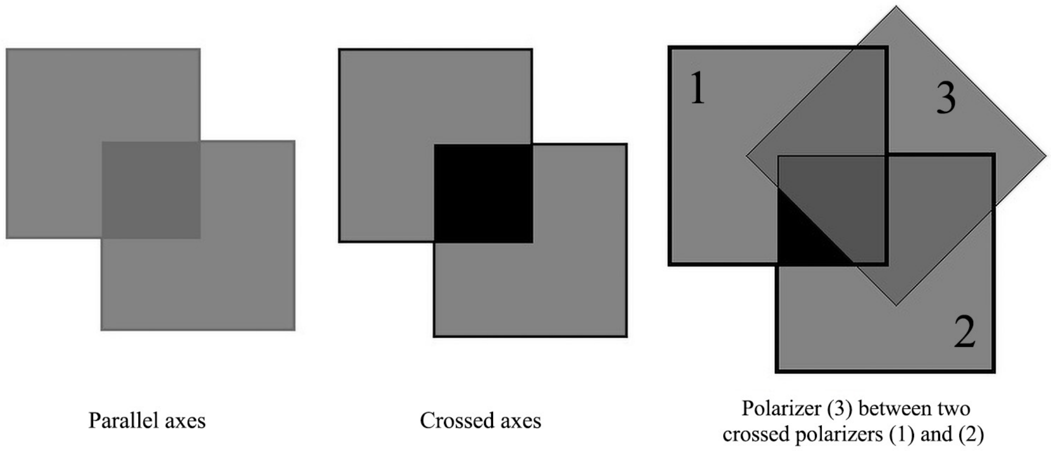


Figure 9.8: The polarized filter experiment

This result shows the peculiarity of the Q-Mechanics world; our logic would lead us to think that less light can pass through by adding one more lens and that it should therefore get darker, but the opposite is the case.

Now that we know what polarization is (and its strange behavior when we deal with photons and quantum mechanics), we will start to explore one of the first applications implemented in quantum mechanics, quantum money.

So, what Wiesner proposed was to create a special banknote in which there are 20 **light traps**, which behave like filters. These traps are oriented so that the polarizations of the photons are directed in four directions, as shown in the following diagram of a 1-dollar note.

The filters can detect (by combining themselves with the serial number on the note) whether the money is real or fake:

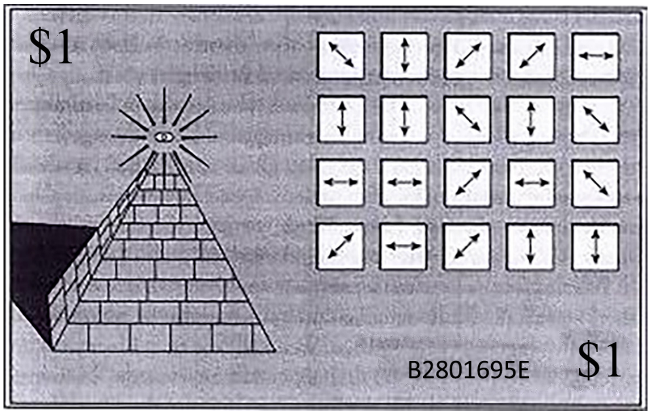


Figure 9.9: Quantum 1-dollar banknote proposed by Wiesner

Only the bank that issues the notes can detect whether they are real or fake. In fact, due to the properties of Q-Mechanics, and in particular because of the indeterminacy principle, if someone wanted to counterfeit a note, it would be impossible to do so.

Let's see why an attacker would find it very hard to counterfeit one of these banknotes. In the preceding diagram, you can see the direction of the trap lights' orientation, but in reality, they are invisible so that nobody can see their orientation. Suppose Eve wants to copy the note; she can copy the serial number on the counterfeited note and try to figure out the traps' orientations.

If Eve decides to use a vertical filer to detect the polarization, she can detect all the vertical photons, and conversely, she will be sure that if the photon is horizontal, it will be blocked. But what about the others? Some of them will pass, but she will not be able to know exactly what polarization they have. So, the only thing Eve can do is randomly decide the polarization of all the photons so that they're different from the vertical and horizontal positions.

But now a problem occurs: if the bank runs a test on the forged banknote, they will know that it's fake. Because of the indeterminacy principle, only the bank knows the corresponding filter that's been selected for a determinate serial number.

Wiesner was always ignored when he talked about his invention of quantum money. It isn't a practicable invention to implement because the cost to set up such a banknote would be prohibitive; but this description introduces our next exploration well, which is the **quantum key distribution** (**QKD**).

Before we move on, just a note about quantum money: in his invention, Weisner intended to give banks the power to detect whether someone had attempted to counterfeit a note, but this concept is flawed. In the real world, the issue does not concern banks: it is much more important that peer-to-peer users have control over judging the authenticity of banknotes. So, in my opinion, if quantum money ever comes out, it will be implemented in some other way.

In the end, Wiesner found someone who listened to him – an old friend who was involved in different research projects and extremely curious about science: Charles Bennett. Charles spoke with Gilles Brassard, a researcher at the time in computer science at Montreal University; they started taking an interest in the problem and found out how to implement QKD, which we will explore next.

QKD: BB84

Let's introduce BB84, the acronym that Charles Bennett and Gilles Brassard developed in 1984, valid for QKD. Now that we have learned about the properties of Q-Mechanics, we can use them to describe a technique for distributing bits (or better, **qubits**) through a quantum channel.

Before we go deeper, let's recap what a **qubit** *is*. We have to refer to the "quantum unit information" that's carried by qubits. Like traditional bits, *(0)* and *(1)*, qubits are mathematical entities subject to calculation and operations.

We will use a **bi-dimensional vectorial complex space of unitary length** to define a qubit. In simpler words, we can say that a qubit is a unitary vector, which is acting inside of a two-dimensional space. So far, we can think of a qubit as a polarized photon, similar to the entanglement experiment we looked at in the previous section. I have already introduced the notation to represent the qubits inside "bra-ket": *|0> and |1>*.

I will not go too deep into the mathematics of Q-Mechanics, but we have to go a bit deeper to analyze some of the properties of qubits. If an element carries some information (like qubits do), it is related to a form of computational process, which in this case is quantum computing. We will return to this concept later when we talk about quantum computers.

To understand the geometrical representation of a qubit, take a look at the following diagram:

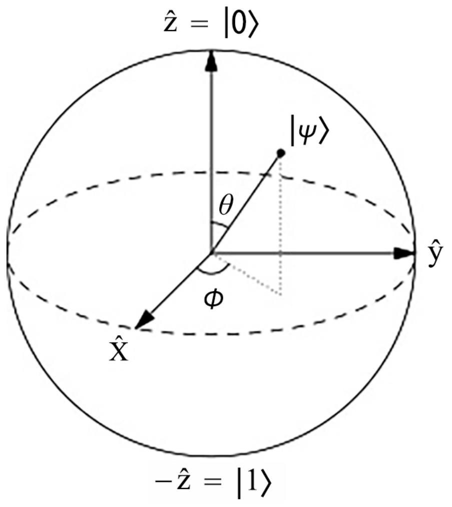


Figure 9.10: Bloch sphere: the fundamental representation of a qubit

The difference between the classical computation we've used so far for all the classical bit operations, based on a Turing machine and Lambda calculus, and quantum computing is substantial. Quantum computing relies on superposition and entanglement properties. Qubits, as you saw when we discussed superposition, can be represented in a simultaneous state of *(0)* and *(1)*, different from the normal bit, which can be only *(0)* or *(1)*.

To better understand the nature of *(0)* and *(1)*, we can briefly consider mathematically representing qubits in the following way:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Figure 9.11: Qubits represented in a vector form

Hence, considering a qubit as a unit vector, it can be represented (as shown in the preceding diagram) as follows:

$$|\psi> = a\,|0> + b\,|1>$$

Here, */a/* and */b/* represent the complex numbers a and b, where the following applies:

$$|a|^2 + |b|^2 = 1$$

Here is the important takeaway: we only know the probability amplitudes, the coefficients IaI and IbI, whose sum of the square's value is *(1)*, but we never know how a photon will be in a superposition state, so the values of the */a/* and */b/* coefficients are unpredictable.

For our scope, that is enough for us to explain the algorithm behind a QKD.

Now that we've provided a brief introduction to the basic concepts of quantum calculus and qubit operations, we are ready to explore the quantum transmission key algorithm.

Step 1: initializing the quantum channel

Alice and Bob need a quantum channel and a classical channel to exchange a message. The quantum channel is a channel where it is possible to send photons through it. Its characteristic is that it must be protected from the external environment. Any correlation with the external world has to be avoided. As we have seen, an isolated environment is necessary for achieving the superposition and entanglement of a particle.

The classical channel is a normal internet or telephonic line where Alice and Bob can exchange messages in the public domain.

We suppose that Eve (the attacker) can eavesdrop on the classical channel and can send and detect photons, just like Alice and Bob can.

Since the interesting part of the system is just the quantum channel, where Alice and Bob exchange the key, we are going to observe what happens in this channel because this is the real innovation of this system. Then, we will make some considerations about transmitting the message through the classical channel.

Step 2: transmitting the photons

Alice starts to transmit a series of bits to Bob. These bits are codified using a base that's randomly chosen for each bit while following these rules.

There are two possible bases for each bit:

$$B1 = \{|\uparrow>, |\rightarrow>\}$$

$$B2 = \{|\nwarrow>, |\nearrow>\}$$

If Alice chooses *B1*, she encodes the following:

$$B1: \quad 0 = |\uparrow> \; and \; 1 = |\rightarrow>$$

If Alice chooses *B2*, then she will encode the following:

$$B2: \quad 0 = |\nwarrow> \; and \; 1 = |\nearrow>$$

Each time Alice transmits a photon, Bob chooses to randomly measure it with the *B1* or *B2* base. So, for each photon that's received from Alice, Bob will note the correspondent bit, *(0)* or *(1)*, related to the base of the measurement that was used.

After Bob completes the measurements, he keeps them a secret. Then, he communicates with Alice through a classical channel and provides the bases (*B1* or *B2*) that were used for measuring each photon, but not their polarization.

Step 3: determining the shared key

After Bob's communication, Alice responds, indicating the correct bases for measuring the photon's polarization. Alice and Bob only keep the bits that they have chosen the same bases for and discard all the others. Since there are only two possible bases, *B1* and *B2*, Alice and Bob will choose the same base for about half of the bits that are transmitted. These bits can be used to formulate the shared key.

A numerical example is as follows.

Suppose that Alice wants to transmit the following bits sequence:

$$[0, 1, 1, 1, 0, 0, 1, 1]$$

The polarizations for each base are as follows:

	0	1
B1	|↑>	|→>
B2	|↖>	|↗>

Alice randomly selects the bases, and then Bob also randomly selects bases to measure with:

Alice's bases	B1	B2	B1	B1	B2	B2	B1	B2
Alice's sequence	0	1	1	1	0	0	1	1
Alice's polarization	|↑>	|↗>	|→>	|→>	|↖>	|↖>	|→>	|↗>

Bob's bases	B2	B2	B2	B1	B2	B1	B1	B2
Bob's polarization	|↖>	|↗>	|↖>	|→>	|↖>	|↑>	|→>	|↗>
Bob's sequence	0	1	0	1	0	0	1	1

Comparing Bob's choices to Alice's, we can see that the second, fourth, fifth, seventh, and eighth polarizations match. Using the bits that correspond to those polarizations, we can establish a shared key:

Correct polarizations	|↗>	|→>	|↖>	|→>	|↗>
Corresponding bits	1	1	0	1	1
[Shared key]	1	1	0	1	1

The following is a representation of this scheme:

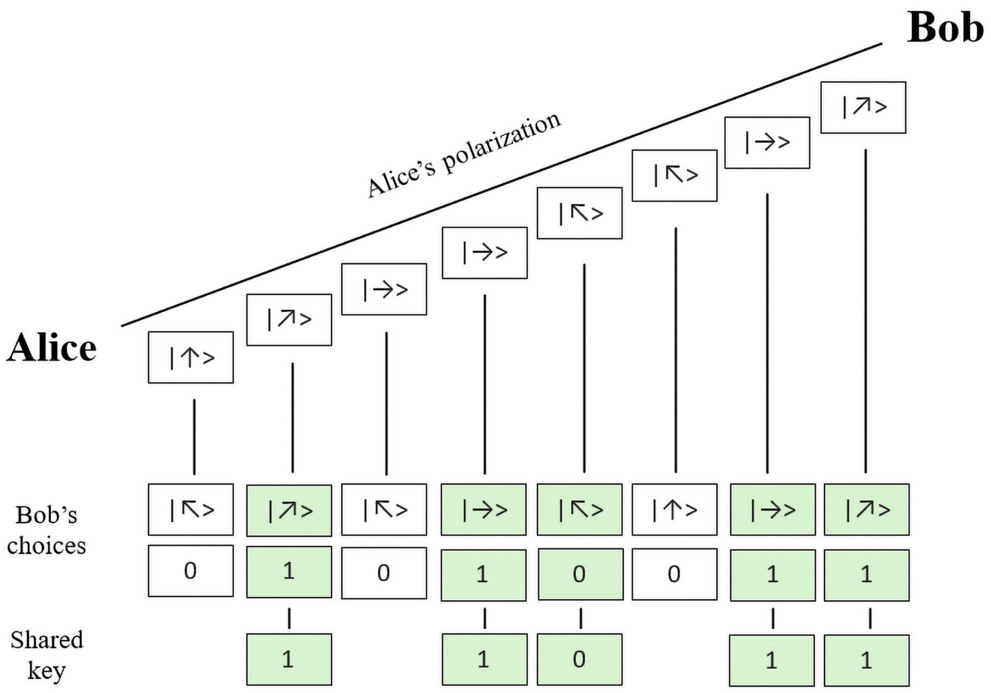


Figure 9.12: Representation of QKD

At this point, we only need to check whether the two keys (from Alice and Bob) are identical.

Alice and Bob don't have to check the entire sequence, but just a part of it to perform the parity bit control. If they check the entire parity sequence via the classical channel, it will be obvious that Eve can spy on the conversation and steal the key. Therefore, they will decide to control only part of the sequence of bits.

Let's say that the key is of the order of 1,100 digits; they will check only 10% of the bits. After the parity control (done by a specific algorithm), they will discard the 100 bits that were processed in the control and will keep all the others. This method lets them know that there were no errors in the transmission and detection process and whether Eve has attempted to observe the transmission in the quantum channel. The reason for this is related to Q-Mechanics' impossibility to observe a photon without forcing it to assume a direction of polarization (the indeterminacy principle). But if Eve measures the photons before Bob and allows the photons to continue for Bob's measurement, she will fail about half of the time. But when the same photon that's being measured by Eve arrives at Bob, he will have only a 25% probability of measuring an incorrect value. Hence, any attempt to intercept augments the error rating. That is why, after the third step, Alice and Bob performing parity control on the errors can also detect Eve's spying.

Now, the question of the millennium: *is it possible for Eve to recover the key?*

In other words, is this algorithm unbreakable?

Let's try to discover the answer in the next section, where we will analyze possible attacks.

Analysis of attack and technical issues

When we first described this algorithm, the assumption was that Eve had the same computational power as Bob and Alice and could spy on the conversations between them. So, if Eve has a quantum computer in her hands and all the instruments to detect the photons' polarization, could she break the algorithm?

Let's see what Eve can do with her incommensurable computational power and the possibility of eavesdropping on the classical and quantum channels.

It looks like she is in the same situation as Bob (receiving photons). When Alice transmits the photons to Bob, even if the transmission comes through a quantum channel, if someone can capture the entire sequence of bits before Bob's selection, it's supposed that they will be able to collect the bits of the key. This could be true if Eve gets 100% of the measurements correct, but that is unthinkable for a key with a 1,000-bit length or more.

Moreover, the attack will be discovered by Alice and Bob, as we will see now.

Let's analyze what can happen. Suppose that Alice transmits a photon with a polarization of /→> and Bob uses a base (*B1*) that's the same as Alice's. If Eve measures the photon with the same base (*B1*), the measurement will not alter the polarization, and Bob will correctly measure the photon's state. But if Eve uses *B2*, then she will measure the two states, /↖>, /↗>, with an equal probability. The photon that will arrive at Bob will only have half the probability of being detected as /→> and correctly measured. Considering the two possible choices Eve has to determine a correct or incorrect measurement, Bob will only have a 25% chance of measuring a correct value.

In other words, each time Eve attempts to measure augments the probability of an error rating occurring in Alice and Bob's communication. Therefore, when Alice and Bob go to check on the parity bits, they will identify the error and understand that an attack is being attempted. In this case, Alice and Bob will repeat the procedure, (hopefully) avoiding the attack again, or changing the channel of communication – for example, switching in a different optical fiber color code from yellow to green.

While analyzing the system, I endorse the thesis that this system is theoretically invulnerable, but I also agree that it is only a partial solution, mainly because this is not a quantum transmission message, but only a QKD. The reason I said it's theoretically invulnerable is that Eve can be detected by Bob and Alice by relying on the indeterminacy property of Heisenberg. However, a man-in-the-middle attack, even if it doesn't recover the key (Eve could attempt to substitute herself with Bob), could block the key being exchanged between Alice and Bob, inducing them to repeat the operation infinite times, or, as we have seen, to switch in a different quantum channel of communication.

In a document from the **National Security Agency** (**NSA**) of the United States, you can read about some weaknesses related to QKD and why it is discouraged that you implement it: `https://www.nsa.gov/Cybersecurity/Quantum-Key-Distribution-QKD-and-Quantum-Cryptography-QC/`.

What is to be understood is whether the NSA's concerns are related to the real possibility of implementing such a system or the possibility that hackers and other bad people could take advantage of this strong system.

In the NSA's document, we note some issues related to the implementation of QKD in a real environment. The NSA's notes are explained as follows:

- **QKD is only a partial solution**: The assumption is that the protocol doesn't provide authentication between the two actors. Therefore, it should need **asymmetric encryption** to identify Alice and Bob (that is, a **digital signature**) by canceling quantum security benefits. This problem could be avoided by using a quantum algorithm of authentication for implementing a quantum digital signature, if possible. Unfortunately, with the current state of the art, the way of identifying a digital signature through a quantum algorithm and the non-repudiation of the message gives only a probabilistic result, far from the deterministic digital signature of the classical algorithms. Another solution is using a MAC (a function for authenticating symmetric algorithms), but again, it may *not* be quantum-resistant in the future.
- **Costs and special equipment**: This problem is concerned with higher costs and implementation problems for users. Moreover, the system is difficult to integrate with the existing network equipment. Anyway, I would comment that the document by the NSA was probably written before the massive advent of optical fiber. Now, our infrastructures are more and more based on optical fiber to carry information. So, this infrastructure is the same used to carry light or photons.
- **Security and validation**: The issues, in this case, are related to the hardware being adapted to implement a quantum system. The specific and sophisticated hardware that's used for QKD can be subject to vulnerabilities. Also, this point could be discussed. In fact, the hardware used now has enormously improved the quality.
- **Risk of denial of service**: This is probably the biggest concern of QKD. As I mentioned previously, Eve can repeat the attack on the quantum channel, sequentially causing Alice and Bob to never be able to determine a shared key. We are studying a new method to overcome this problem using different optical color fibers to switch automatically in case of dynamic detection of an anomaly in the quantum channel.

Stay updated by visiting our fantastic Quantum Cryptography Laboratory Center! Its websites are `www.quantumlab.science` and `www.quantumlab.educational`.

Note that the Q-Lab will be operative starting from November 2024.

As we have seen, the QKD method is not unerring. There is a remote possibility that someone, in practice, could break the system.

Important note

QKD is just a part of the cryptographic transmission algorithm; the second part of the system is the encryption algorithm that's used to send the message, *[M]*.

This is a fundamental part of Q-Cryptography's evolution because, to be complete, the system should guarantee the transmission of the message, *[M]*, through a quantum channel, but that needs another quantum algorithm. Moreover, the quantum authentication protocol remains unsolved.

We can say that QKD could be compared to **Diffie–Hellmann**, as **RSA** could be compared to a **quantum message system transmission**.

There are various choices we can make when it comes to transmitting messages in classical cryptography; we can use **Vernam** (as we saw in *Chapter 1*, *Deep Dive into Cryptography*), which, combined with a random key (QKD), is theoretically secure, or another symmetric algorithm. But again, we don't rely on a fully quantum-based encryption system, which is autonomous and independent of classical mathematic problems.

Returning to the QKD algorithm, we have to find out why this algorithm is theoretically impossible to break, even with a quantum computer.

If an attacker decrypted a key generated with Q-Cryptography, the result of this action would probably be devastating for the entirety of Q-Mechanics. This is because one of the axioms that Q-Mechanics bases its strength on would be inconsistent: we are talking about the indeterminacy principle. If something like this happened, then the complete structure of Q-Mechanics would probably be involved in a revision process that would shock the pillars of the theory and all its corresponding physical and philosophical implications.

To establish this concept, I propose a theory: suppose that, in the future, someone will discover that the photons passing between the point of Alice's transmission and the point of Bob's measurement are not truly random but follow a pre-determined scheme. Suppose that this scheme is π or a Fibonacci series. In this case, the bits are effectively random, but their randomicity will be known so that Eve can forecast what the polarized scheme to adopt is.

What happens, for example, if we recognize that a sequence of digits such as "1415" is included in π?

As shown below, π is a sequence composed of an infinite number of digits that follow the number 3, starting with "1415".... after the decimal point:

π = 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190...

We can go on following the π sequence and recovering all the other digits after *1415*: *92653589*...

Undoubtedly, I agree with the NSA's report when they said that QKD is an incomplete algorithm. It misses the second part of the algorithm, which completes the encryption of the message *[M]* into the quantum channel. By using BB84, this stage is devolved to classical cryptography because after exchanging the key (in the quantum channel), we must encrypt the message using a classic encryption algorithm. This transmission has two issues:

1. As said, it's classical, even if theoretically secure.
2. With a classical algorithm, we are not able to detect the intrusion of Eve. In contrast, we can do this using a quantum algorithm transmission.

As I have always been against hybrid schemes, in 2023 I designed a new quantum encryption algorithm valid for quantum message system transmission, called **QTM23**. It will hopefully become a standard for the next Q-Cryptography transmissions.

In *Cryptolab*, we are now in the prototyping phase. On June 19, 2024, we transmitted a "HELLO WORLD!" message through a quantum channel with QTM23, using the same hardware we used to experiment with the BB84. Using the names we have grown accustomed to throughout this book, Bob received the message about 6 minutes after Alice transmitted it. We are currently using mechanical motorized beam detectors to shoot photons one by one, which is technically very slow. In the future, we are aiming to improve the speed using LED beam detectors.

Due to patent issues, it is not possible to explain the algorithm here. I hope to be able to update this section with more details on this innovation soon.

Now that we have analyzed the Q-Cryptography key distribution, another challenge awaits: exploring quantum computing and what it will be able to do when its power increases.

Quantum computing

In this section, I will talk about **quantum computing** and **quantum computers**. The difference between the two is that quantum computing is the computational power that's expressed by a quantum system, while a quantum computer is the system's physical implementation, which is composed of hardware and software. The first idea of a quantum computer was originally proposed by Richard Feynman in 1982; then, in 1985, David Deutsch formulated the first theoretical model of it. This field has gone through a big evolution in recent decades; some private companies began to take action and experiment with new models of quantum computers. Some of them, such as **D-Waves** and **Righetti Computing**, raised millions of dollars for the research and development of these machines and their relative software:

Figure 9.13: D-Waves hybrid quantum computer

As you should already know, normal computers work with bits, while quantum computers work with qubits. As we have also already seen, a qubit has a particular characteristic in that it can be simultaneously in the state of *(0)* and *(1)*. Just this information by itself suggests a greater capability of computation concerning a normal computer.

A normal computer works by performing each operation one by one until it reaches the result. Conversely, a quantum computer can perform more operations at the same time. Moreover, many qubits can be linked together in a state of entanglement.

This phenomenon creates a new state of superposition over the multiple combinations of the entangled qubits, elevating the grade of computation of a quantum computer exponentially compared to a normal one. So, we are not only talking about a parallelization process of computation but also an exponential computation process when augmenting the number of qubits in the game. Therefore, to give you an idea of the computational power that's created by a quantum computer, *n* qubits can generate a capability to elaborate information 2^n times compared to classical bits.

Let's consider the preceding sentence. In a state of superposition or entanglement, 8 qubits, for instance, can represent any number between *0* and *255* simultaneously. This is because $2^8 = 256$, so we have 256 possible configurations each time, which, expressed in binary notation, looks like this:

$$0 = (0)^2 \rightarrow 255 = (11111111)^2$$

We can understand the supremacy of a quantum computer, especially if we deal with two specific problems of extreme interest to cryptography: the factorization problem and searching on big data.

These two arguments are considered very hard to solve for a classical computer; we have seen, for instance, the RSA algorithm and other cryptographic algorithms relying on factorization to generate a one-way function that's able to determine a secured cryptogram. However, the community's recurring question is, *how long will classical cryptography resist when a quantum computer reaches enough qubits to perform operations that are considered unthinkable?*

We will try to answer this question after exploring the potential of a quantum computer and seeing what we can do with one by projecting one adapted to our scope.

As we mentioned previously, a quantum computer can simultaneously perform all the state bases of a linear combination. Effectively, in this sense, a quantum computer can be considered an enormous parallel machine that's able to calculate huge states of combinations in parallel.

For example, to understand what a quantum computer can do, let's take three particles: the first of orientation *1* and the last two with an orientation of *0* (relative to some base). We can represent this as follows:

$$|100>$$

The quantum computer can take */100>* as input and give some output, but at the same time, it could take a *normalized combination* of the three particles with a state base as input, as follows:

$$1/\sqrt{3}(|100> + |100> + |100>)$$

As a result, it gives an output in only one operation as it computes only the state base.

After all, the quantum computer doesn't know whether a photon is in one state base or multiple combinations of states until a measurement occurs (the indeterminacy principle).

This ability to elaborate combinations of linear states simultaneously determines the quantum supremacy of quantum computers.

So, while a classical computer (even the most powerful supercomputer on earth) requires a string of bits as input and returns a string of bits as output (even if it's parallelizing billions of operations), it will not be able to perform a function that accepts the following sum as input:

$$1/C \sum |x>$$

Here, *C* is a normalization factor of all the possible states of the input. The function would return the following as output:

$$1/C \sum |x, f(x)>$$

Here, */x, f(x)>* is a succession of bits longer than */x/*, which represents both */x/* and *f(x)*.

To understand this concept, you can think of having a sum of bits, *(C)*, and in this sum, all the possible states of *(m)* qubits in a superposition are hidden. So, we can say that *(C)* represents the number of all the states that the qubits are in at the same time.

You can imagine how fantastic that could be! But still, there is a problem: the measurement. We have seen that when we measure the quantum computation, this measurement *freezes* the qubits in only one state, which is governed by randomness. Therefore, we can force the system *to appear* in a state, as follows:

$$|x0, f(x0)>$$

This is for some *x0* value that's randomly chosen, so all the other states are going to be destroyed.

In this case, the ability to program a quantum computer makes it possible to create a quantum computation that provides a very high-probability result of getting the desired output.

To understand the process of implementing a quantum algorithm, let's look at a candidate algorithm that could wipe out a great part of classical cryptography: Shor's algorithm.

Shor's algorithm

In an interview with David Deutsch (considered one of the fathers of quantum computing), the interviewer posed a question about the philosophy of superposition, asking David, *"In what way does the quantum computing community give credit to the hypothesis of a multiverse?"*

David's answer refers to Q-Cryptography and, in particular, he tries to demonstrate the existence of the multiverse through the factorization problem. I will try to paraphrase David's answer in the following paragraph:

Imagine deciding to factorize an integer number of 10,000 digits, a product of two big prime numbers. No classical computer can express this number as the product of its prime factors. Even if we take all the matter contained in the universe and transform it into a supercomputer, which starts to work for a time long, such as the universe's time, this instrument will not be able to scratch the surface of the factorization problem. Instead, a quantum computer should be able to solve this problem in a few minutes or seconds. How is this possible?

Whoever is not a solipsist has to admit that the answer is linked to some physics process. We know that not enough power exists in the universe to calculate and obtain an answer, so something has to happen compared to what we can see directly. At this point, we have already accepted the structure of the multiverse.

This digression into *quantum computational power emphasizes the potential of quantum computers*. Let's dive into Shor's algorithm mathematically and logically, as this algorithm is the true representation of what happens in a quantum computer at work.

Hypothesis and thesis

The *hypothesis*, of Fermat's Last Theorem, is as follows: say you find two (non-trivial) values, *(a)* and *(r)*, so the following applies:

$$a^r \equiv 1 \ (mod\ n)$$

Here, there is a good possibility of factorizing *(n)*.

We start to randomly choose a number, *(a)*, and consider the following sequence of numbers:

$$1, a, a^2, a^3, \dots, a^n \ (mod\ n)$$

The *thesis* is that we can find a period for this sequence that we call *(r)*, which will be repeated every (*r*) time when we elevate *(a)* to *(r)* so that we will have solved the following equation for *(r)*, which always gives *(1)* as a result:

$$a^r \equiv 1 \ (mod\ n)$$

Therefore, we want to program our quantum computer to find an output, *(r)*, to be able to factorize *(n) with a high probability*.

Step 1: initializing the qubits

We choose a number, *(m)*, so that the following applies:

$$n^2 \leq 2^m < 2^n$$

Supposing we choose *m = 9*, this means that we have 9 bits initialized with their state all in *(0)*, and we mathematically represent it as follows:

$$m = 9: |000000000>$$

Changing the axes of the bits, we can transform the first bit into a linear combination of */0>* and */1>*, which gives us the following:

$$1/\sqrt{2}\,(|000000000> + |100000000>)$$

By performing a similar transformation on every single bit of *(m)* inside *bra-ket* until the *m-bit to obtain the quantum state*, we get a sequence like this:

$$1/\sqrt{2^m}(|000000000> \ + |000000001> \ + |000000010> \ + \cdots + |111111111>)$$

Remember that *m = 9*, so 2^m *= 512*; we will have 512 multiple combinations of bits, given by the first, */000000000 = 0*, and the last, */*2^m*-1> = /111111111> = 511* (in decimal notation).

In a sum like this, all the possible states of the *m* qubits are overlying in a superposition. Now, for simplicity regarding notation, we will rewrite the *m* qubits in cardinal numbers:

$$1/\sqrt{2^m}(|0> + \ |1> + \ |2> + \cdots + |2^{m-1}>)$$

The preceding functions are just the same as the previous ones but are represented by cardinal numbers.

Step 2: choosing the random number, (a)

Now, we must choose a random number (a) so that the following applies:

$$1 < a < n$$

Quantum computing computes the following function:

$$fx \equiv a^x \ (mod\ n)$$

It returns the following:

$$1/\sqrt{2^m}(|0, a^0 \ (mod\ n) > \ + |1, a^1 \ (mod\ n) > \ + |2, a^2 \ (mod\ n) > \ + \cdots + |2^{m-1}, a^{2m-1} \ (mod\ n) >)$$

When visualizing this function, you can lose the sense of representation because it appears rather complex; so, to simplify it, forget the modulus, *n*, that's repeated for each operation and rewrite the previous functions like so:

$$1/\sqrt{2^m}(|0, a^0 > \ + |1, a^1 > \ + |2, a^2 > \ + \cdots + |2^{m-1}, a^{2m-1} >)$$

Again, consider that all the operations are in *(mod n)*, but focus on the sense of the function: it says that we have to elevate *(a)* to all the (2^{m-1}) states of the *m* qubits.

However, so far, the *state* of this representation doesn't provide any more information compared to a classical computer.

Now, it's time to look at an example to help you understand what we are doing.

Suppose that *n = 21* (a very easy number to factorize); however, the size of the number is not important now but the method is.

Here, we choose *a = 11* randomly.

So, we calculate the values of *11^x (mod 21)* that have been transposed into the function:

$$1/\sqrt{512}\ (|0, 1 > \ + |1, 11 > \ + |2, 16 > \ + |3, 8 > \ + |4, 4 > \ + |5, 2 > \ + |6, 1 > \ + |7, 11 > \ + |8, 16 > \ + |9, 8 > \ + |10, 4 > \ + |11, 2 > \ + |12, 1 > \ + |13, 11 > \ + |14, 16 > \ + |15, 8 > \ + |16, 4 > \ + |17, 2 > \cdots + \ |508, 4 > \ + |509, 2 > \ + |510, 1 > \ + |511, 11 >)$$

As you can see, this is a development of the entire sequence, where *1/√512 = 1/C* and *C* is the number of states of *m = 2^9*.

Each qubit is composed of a sequence number, *(a^x (mod 21))*. Take the following example:

$$|1, 11 > \equiv 11^1\ (mod\ 21) = 11$$

Step 3: quantum measurement

If we measure only the second part of the sequence, a^x, then the function collapses in a state that we cannot control. However, this measurement is the essence of the system. In fact, because of this measurement, the entire sequence collapses for some random base, *(x0)*, so the following applies:

$$|x0, a^{x0} >$$

Now, it should be clearer to you that we can force the system *to appear* in a state, as follows:

$$|x0, f(x0) >$$

So, after the measurement, the state of the system is fixed (in other words, a number appears) in only one base ($x0$) that we suppose to be as follows:

$$x0 = 2$$

Forgetting all the other values of a^x (that we have previously calculated) and picking up only the ones that have a value equal to 2, we will extract the following highlighted values:

$$1/\sqrt{512}\,(|0,1 > +|1,11 > +|2,16 > +|3,8 > +|4,4 > +|\mathbf{5},\mathbf{2} > +|6,1 > +|7,11 > +|8,16 > +|9,8 > +|10,4 > +|\mathbf{11},\mathbf{2} > +|12,1 > +|13,11 > +|14,16 > +|15,8 > +|16,4 > +|\mathbf{17},\mathbf{2} > \cdots + |508,4 > +|\mathbf{509},\mathbf{2} > +|510,1 > +|511,11 >)$$

To simplify the notation, we must discard the second part of the qubit (2) because it is superfluous:

$$1/\sqrt{85}\,(|5 > +|11 > +|17 > \cdots + |509 >)$$

Here, *85* stands for the number of m qubits selected.

Now, if we take a measurement, we will find a value, x, so that the following applies:

$$11^x \equiv 2\ (mod\ 21)$$

Unfortunately, this information is not useful for us. But don't lose hope.

Step 4: finding the right candidate, (r)

Recall that we have programmed our quantum computer with a specific scope: to find the output in the system that can factorize *(n)*. To reach our goal, we have to find the value of *(r)* so that the following applies:

$$a^r \equiv 1\ (mod\ n)$$

Generally, when we deal with functions such as a^x *(mod n)*, we know that the values of *(x)* are periodic, with a period of *(r)* where $a^r = 1\ (mod\ n)$.

So, recalling the previous function in which the selected elements appear, we have the following:

$$|5> \ +|11> \ +|17> \cdots + \ |509>$$

In this example, $r = 6$ is the period of the function.

Using the result $r = 6$, we can verify whether the following applies:

$$11^r \equiv 1\ (mod\ 21)$$

$$11^6 \equiv 1\ (mod\ 21)$$

The period of the function is not always easy to work out. There is a mathematical way to discover the period of a function, using the **quantum Fourier transform** (**QFT**). We will learn about that in the next section.

Alternatively, if you want to continue looking at the demonstration of Shor's algorithm, you can skip the next section and go directly to the last step of Shor's, coming back to QFT on your second read.

QFT

Let's discover a little bit more about the **Fourier transform** (**FT**) and **QFT.**

FT is a mathematical function. It can be intuitively thought of as a musical chord in terms of volume and frequency. The *FT* can transform an original function into another function, representing the amount of frequency present in the original function. The FT depends on the spatial or temporal frequency and is referred to as a **time domain**.

For example, suppose we assume that a function, *f*, is a succession of numbers made up of **periods**, represented by the frequency in which the succession takes place. We can represent these periods with curves called **harmonics**.

An example of the *FT* is shown in the following graph, showing the frequency that was detected in the harmonics:

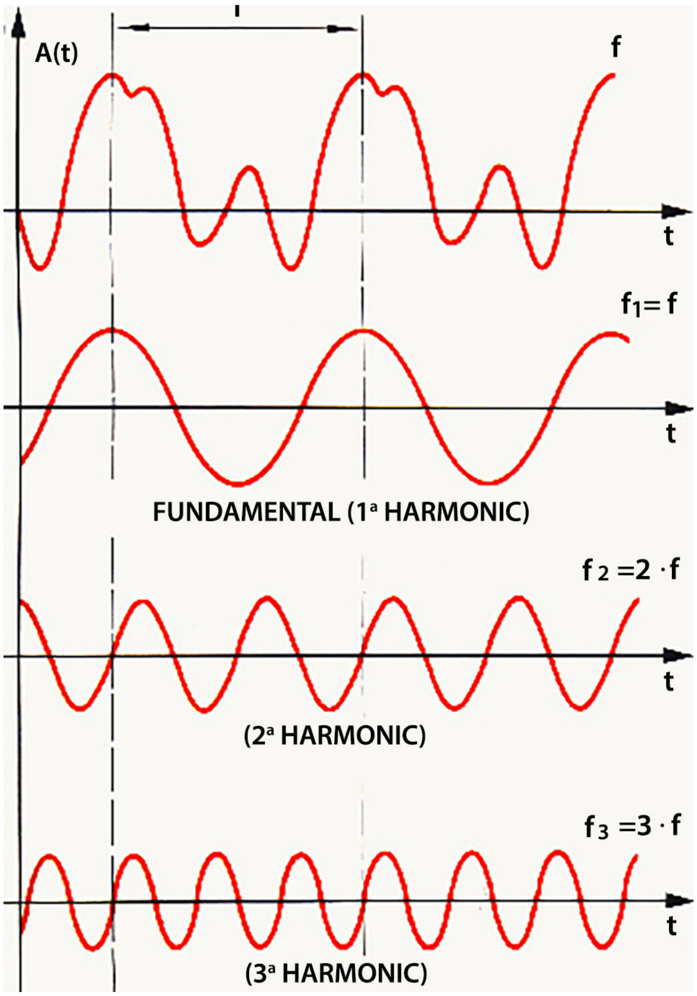


Figure 9.14: Fourier transform

There are four subplots, representing the function, *f*, and three different harmonics. We can see a period captured within each of these subplots with the dotted lines. Remember that the curves within that period are a representation of a sequence of numbers. The shape of the harmonics depends on the frequency at which that sequence recurs.

To understand how this works from another angle, let's look at a numerical example of FT taking an arithmetic succession of numbers, like so:

$$1, 3, 7, 2, 1, 3, 7, 2$$

As you can see, we have the first four numbers: $1, 3, 7, 2$, which get repeated two times. In other words, we can break the succession into two parts:

1,3,7,2 | 1,3,7,2

Important note

The | symbol represents, in this case, splitting the succession into two parts.

Here, we can say that this succession has a length of 8 and a period of 4. The length, divided by the period, gives us the frequency (*f*):

$$f = 8/4 = 2$$

In this case, *f = 2* is the number of times that the succession is repeated.

Now, we can announce the general rule of FT. Suppose we have a succession that's *2*^{m} in length for any integer, *m*:

$$[a0, a1, \dots, a2^{m-1}]$$

We can define the FT with the following formula:

$$FT(X) = 1/\sqrt{2^m} \sum_{c=0}^{2^{m-1}} e^{(2\pi icx/2^m)}\ c$$

We can define the following part of the formula with *j*:

$$j = e^{(2\pi icx/2^m)}$$

The *x* parameter runs in the following range:

$$0 \leq x < 2^m$$

With the FT formula, you can find the frequency of the succession while using determinate parameters as input. If we suppose, for example, *cx = 1* and *m = 3*, then we have the following substitution:

$$j = e^{(2\pi i/8)}$$

By substituting the different values of *x* in the equation with *j*, we obtain the following result by exploding the formula for *FT(1)*:

$$\sqrt{8}FT(1) = 1 + 3j + 7j^2 + 2j^3 + j^4 + 3j^5 + 7j^6 + 2j^7 = 0$$

Since all the terms of the succession get eliminated, we have the following:

- *FT(X0) with* $x = 0 \dashrightarrow e^0 = 1$, which is the result of *FT(X0) = 1*
- *FT(X4) with* $x = 4 \dashrightarrow e^{\pi i} = -1$, which is the result of *FT(X4) = -1*

So long as $j^4 = -1$, all the terms will be deleted and we will obtain *FT(1) = 0*.

Instead, for *FT(0)*, the result of the numerator addition corresponds to the sum of all the terms of the sequence:

$$1 + 3 + 7 \ldots + 2 = 26$$

Continuing, we have the following:

$$\begin{array}{rl} FT(0) = 26/\sqrt{8} & FT(2) = (-12 + 2i)/\sqrt{8} \\ FT(4) = 6/\sqrt{8} & FT(6) = (-12 - 2i)/\sqrt{8} \end{array}$$

For all the other terms, *FT(1) = FT(3) = FT(5) = FT(7) = 0*.

In other words, the FT confirms what we have deducted since the beginning of this example: that is, in the succession 1, 3, 7, 2, 1, 3, 7, 2, the peaks of the function correspond with the non-zero values of *(FT)* that are multiples of 2: the frequency.

This result will be very useful in Shor's algorithm to discover the period of the function, as we will see shortly.

But we want to program our algorithm in a quantum computer, so we need a version of FT that's been adapted for quantum algorithms.

QFT is what we need to detect the frequencies that we use to find the period (*r*) in Shor's algorithm. It is defined on a base state of */X>* with $(or \le x < 2^m)$:

$$QFT(|X>) = 1\Big/\sqrt{2^m}\sum_{c=0}^{2^{m-1}} e^{(2\pi icx/2^m)}\,|c>$$

As you can see, the FT (shown previously in this chapter) and QFT (shown here) are very similar. The difference is that QFT is acting in a quantum environment, so the *c* parameter of FT now appears in a quantum state, */c>*, and the same is valid for */X>*.

Returning to our example of Shor's algorithm, we can apply the QFT to it to discover the frequency:

$$QFT\left(1/\sqrt{85}\ (|5> \ + |11> + |17> \cdots + |509>)\right)$$

We obtain the following sum:

$$1/\sqrt{85}\sum g(c)\ |c>$$

For *c*, this runs from 0 to 511:

$$g(c) = 1\Big/\sqrt{512}\sum_{0\leq x\leq 512} e^{(2\pi icx/512)}$$

This is the FT of the following succession:

$$0,0,0,0,0,1,0,0,0,0,0,1.\ldots.0,0,0,0,0,1,0,0$$

By applying a break symbol to the succession, we can visualize the frequency:

$$0,0,0,0,0,1\ |0,0,0,0,0,1\ |.\ldots.0,0,0,0,0,1\ |,0,0$$

As you can see, the frequency of this succession is around *512/6 ~ 85*.

In this sinusoidal function, which is given by waves in which there are high and low points, the picks (highest points of interest) of the function correspond with the following:

$$c\ =\ 0, 85, 171, 256, 314, 427$$

We expect that *427* is a multiple of the frequency we are looking for:

$$427{\sim}jf0$$

This applies so long as the following is true:

$$427/512 \sim 5/6$$

So, we expect that *r = 6* is the period.

Important note

I didn't show all the mathematical passages that bring us to the result of *r = 6*, but the intention is to give you a demo of the possibility of applying QFT to this algorithm. For a deeper analysis of QFT, go to the following session of the IBM Quantum Lab project: `https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html`.

Now that we know what QFT does and how to use it in Shor's algorithm, let's perform factorization through a quantum algorithm.

Step 5: factorizing (n)

The last step of Shor's algorithm uses the **greatest common divisor** (**GCD**) to calculate whether the candidate *(r)* gives back the two factors of *(n)*.

As you probably remember from high school, the GCD is the greatest positive common divisor between two numbers. It can also be used to reduce a fraction to its smallest possible divisors.

Regarding the Mathematica software, there's a very useful reduction function that is implemented when you divide by two integers.

In our case, if *r = 6* and *n = 21*, we just divide the two numbers:

- $In\,[01]:=\ GCD\,[21,6]$
- $Out\,[01]:=\ 3$

Here, I have told Mathematica to perform *GCD[21,6]*, finding the number *(3)*: the GCD between *(21)* and *(6)*.

Now, by reducing the fraction (the operation consists of dividing both *21* and *6* by *3*), we get *7* as the second output:

- $In\,[02]:=\ 21/6$
- $Out\,[02]:=\ 7/2$

Therefore, *21 = 7*3*.

We got the solution!

Notes on Shor's algorithm

Peter Shor presented this algorithm when the quantum computer hadn't been experimented with and built up in practice. Here, I provided an example of how it could be possible to perform a factorization in a theoretical way on a quantum computer, but at the moment, no machine exists that can perform this algorithm in a scalable way.

Moreover, Shor's algorithm is not deterministic. Still, it gives a probabilistic grade to find the factors, even if, generally, after a few steps (if the first candidate that's found is discarded), it will be possible to find a good candidate for the factorization of (n).

From the opposite point of view, if a scalable quantum computer were to be implemented and ready to run this algorithm in the future, it would probably be the end of classical cryptography.

But now, it's time to answer our initial question about quantum computers, reformulated appropriately: what are the consequences of cryptography when the quantum computer reaches enough qubits to perform Shor's algorithm?

Let's discuss this topic in the final section of this chapter, which is dedicated to **post-Q-Cryptography**.

Post-Q-Cryptography

It is useful to point out that post-Q-Cryptography has nothing to do with Q-Cryptography, except for the fact that it could be considered resistant to quantum computing (maybe it is only a chimera). When we talk about post-Q-Cryptography, we refer to classical algorithm candidates being resistant to quantum computing.

If quantum computing were to raise its power, expressed in a satisfying number of qubits, and were implemented on appropriate hardware able to perform the quantum computation, many of the algorithms that have been discussed in this book would be breached. However, nowadays, a quantum computer occupies an entire room of space and works through hardware made by several components (most of them prepared to keep the qubits in a state of entanglement at frozen temperatures). As soon as the dimensions of the hardware are reduced and its calculation power increases, most of the following algorithms would probably fail in terms of timing, which could swing between 1 hour and a few days: RSA and Diffie–Hellman, Schnorr protocols, and most of the zero-knowledge protocols – even elliptic curves; it is supposed that most elliptic curve implementations would be breakable.

Based on a document provided by the **European Commission for Cybersecurity** (**ENISA**) released in February 2021, some categories of algorithms could support the shock wave of quantum power computation.

In this study, the commission commented on the advent of quantum computing:

> *It is thus important to have replacements in place well in advance. What makes matters worse is that any encrypted communication intercepted today can be decrypted by the attacker as soon as he has access to a large quantum computer, whether in 5, 10, or 20 years from now; an attack known as retrospective decryption.*

The categories of algorithms that are reputed to be resilient to quantum computing, many of them previously evaluated by NIST, are as follows:

- **AES256**: This is considered a good post-quantum candidate: we saw this algorithm in *Chapter 3, Asymmetric Encryption Algorithms*.
- **Hash functions**: We saw these functions in *Chapter 4, Hash Functions and Digital Signatures*.
- **Code-based cryptography**: This uses the technique of error-correction code, based on a proposal by *McEliece*.
- **Isogeny-based cryptography**: This is a kind of cryptography based on adding points to elliptic curves over finite fields.
- **Lattice-based cryptography**: Among these algorithms, the NTRU algorithm is probably one of the best candidates that resists quantum computing.

All these categories are supposed to be quantum-resistant, based on evaluations given on mathematics and cryptoanalysis of the mentioned systems. However, some skeptical thoughts are spreading among the community, most of which are often calmed down by announcements about minimizing problems.

Summary

In this chapter, we analyzed Q-Cryptography. After introducing the basic principles of Q-Mechanics and the fundamental bases of this branch of science, we took a deep dive into QKD.

After that, we explored the potential of quantum computing and Shor's algorithm. This algorithm is a candidate for wiping out a great part of classical cryptography when it comes to implementing a quantum computer that will get enough qubits to boost its power against the factorization problem.

Finally, we saw the candidate algorithms for so-called post-Q-Cryptography; among them, we saw **AES** (**Advanced Encryption Standard**), **SHA** (**Secure Hash Algorithm**), and **NTRU** (**N-th Degree Truncated Polynomial Ring Units**).

With that, you have learned what Q-Cryptography is and how it is implemented. You have also learned what quantum computing is and, in particular, became familiar with a very disruptive algorithm that you can implement in a quantum machine: Shor's algorithm.

Finally, you learned which algorithms are quantum-resistant to the power of the quantum computer and why.

These topics and the concepts that were explained in this chapter are essential because the future of cryptography will rely on them.

Now that you have learned about the fundamentals of Q-Cryptography and most of the principal cryptography algorithms that have been produced until now, it's time to move on to *Chapter 10*, where we explore another important quantum algorithm: Grover's quantum search algorithm. To unpack this algorithm, we will approach the basics of quantum programming.

10

Quantum Search Algorithms and Quantum Computing

We will now discuss a very intriguing quantum algorithm that demonstrates how powerful **quantum search** could be on an unstructured dataset.

Grover's algorithm has potentially many applications, ranging from cryptography to artificial intelligence. In cryptography, it could, theoretically, perhaps in the future, be used to break the symmetric keys of AES or SHA using a large-scale quantum computer. Its principal limitation is that it can only provide a quadratic speedup and not an exponential speedup. So, we can only consider Grover's algorithm to be an early algorithm in the era of quantum computing. I am confident that in the future, a more efficient quantum searching algorithm will be discovered that will improve quantum search's disruptive power to devastate classical cryptography.

In this chapter, we will cover the following topics:

- An overview of Grover's algorithm
- Classical information vs quantum information
- Elements of quantum programming with quantum circuits and gates
- Deep dive into Grover's algorithm

An overview of Grover's algorithm

To illustrate the impact of quantum search on classical cryptography, imagine you have a symmetric algorithm with millions of millions of keys, and you are trying to find the right one to decrypt the ciphertext. Quantum search has the potential to find the right key in seconds or minutes, instead of days or even years!

Grover's algorithm could brute force a 128-bit symmetric key in roughly 264 iterations or a 256-bit key in 2128 iterations. These iterations require a considerable number of qubits, which nowadays is only theoretically possible and not yet feasible.

The quantum search algorithm introduced by Lov Grover in 1996 is important for cryptography, but principally, it will be your introduction to the world of programming algorithms in quantum computing.

Searching for an instance in an unstructured dataset is similar to finding the correct decryption key in a set of random keys.

However, this kind of unstructured search involves various types of problems:

- **Optimization**: Finding the best path to reach a destination between points *A* and *B* visiting many sub-points – *C*, *D*, *E*, *F*... – along the path. For example, this is the kind of problem UPS and FedEx need to solve when optimizing routes through a city.
- **Identification**: Machine learning classification algorithms. In short, classification is a form of **pattern recognition**. For example, a machine learning algorithm could recognize and classify emails as "spam" or "not spam."
- **One-way functions**: We have seen examples of this problem throughout the book, such as hash functions, discrete logarithm problems, and factorization problems, where there is a solution that is hard to find but easy to verify. The digital signature, for example, is emblematic of this: it is possible to verify your identity, but it is very difficult to falsify the signature because a hard mathematical problem supports it. Remember that hash functions, just like all other cryptographic algorithms, are made up of keys, so if we are able to search inside a database of keys, we can brute force these functions. Another case could be discovering a discrete logarithm inside a function such as Diffie-Hellman or factoring large numbers composed of primes (for example, we looked at the RSA algorithm in *Chapter 3*, *Asymmetric Encryption Algorithms*). But for these two categories of problems, in my opinion, Shor's algorithm (we looked at Shor's algorithm in *Chapter 9*, *Quantum Cryptography*) works better than Grover's. In this category, we also have Sudoku solvers.

In general, Grover's algorithm is suitable for all problems related to random search and brute force. However, in order to completely understand the algorithm, we need to deep dive into quantum computing, and hopefully, you will become passionate about quantum programming.

In Grover's original paper related to his quantum search algorithm, we find the following introduction:

> *"Imagine a phone directory containing N names arranged in completely random order. In order to find someone's phone number with a probability of 1/2, any classical algorithm (whether deterministic or probabilistic) will need to look at a minimum of N/2 names. Quantum mechanical systems can be in a superposition of states and simultaneously examine multiple names. By properly adjusting the phases of various operations, successful computations reinforce each other while others interfere randomly. As a result, the desired phone number can be obtained in approximately the square root of N steps."*

Important note

A phone directory is not the best case to choose as an example, because usually it is alphabetized and not random. But to stay on the path indicated by Grover, I will take this as an example.

There are some keywords in the text that will help us to better understand the problem:

- Completely random order
- Probability
- Superposition

First of all, it's clear that in this example, we are struggling with a dataset of **completely random instances**. Our objective is to find the right name in a large list of unordered names. Not pre-ordered in this case means not in alphabetical order. Personally, as I said, I do not completely agree with Grover's example about the names in a phone directory because usually the names are pre-ordered and it's easy to alphabetize them with a classical computer. It might be useful to think of a big dataset of keys that needs to be scanned to find the one that decrypts the ciphertext, or the solution to a complicated equation that is able to crack an asymmetric algorithm.

Also, Grover's introduction puts its focus on the **probability** of finding an instance in this unstructured database. So, we are trying to increase the probability of finding the target instance using a quantum system. In a classical system, we have a 50% chance of finding a number in a dataset of random numbers by exploring the first half of the content, and an equal probability of 50% if we start exploring the second half of the content. Obviously, if you are very lucky you will find the instance on your first try, but the probability of finding the item is still 50%. We will see that the probability of finding the instance increases quadratically using this quantum algorithm.

Grover's introduction tells us that we are able to increase the probability of finding the instance using quantum computing. We can put the system in a **superposition** and simultaneously examine multiple instances of the register (the register can be made of names, numbers, or keys, it doesn't matter).

We will see how to obtain this effect and exactly what it means to put qubits into superposition. Remember that we learned about both qubits and superposition in *Chapter 9*, *Quantum Cryptography*.

Finally, by properly adjusting the phases of various operations, we determine with a high probability of success how to find the instance. It will be approximately the square root of the number of steps N required in a classical system. You will find in mathematics the notation *O(N)*, which is called **big O notation**; the reduction in the number of steps in this notation is *O(√N)*. This notation denotes the upper bound of time/memory used by an algorithm to carry out a task.

In a simple example of choosing a random value between 1 and 100, in the worst case, there are 99 attempts in the classical approach. This would become the square root of 100, or 10 *circuits*, in the quantum case. We will see what a circuit is and what it means to run a circuit in quantum computing. As you can see, we will have a quadratic speedup if we use Grover's algorithm compared with a classical search.

We will see how this fantastic goal can be achieved, but before that, to explore the algorithm, we need to go deeper into quantum computing.

Elements of quantum programming: quantum information and circuits

If we want to start to program a quantum computer, we need to be familiar with quantum information and quantum circuits and the mathematics behind them.

However, this book doesn't pretend to be an exhaustive guide to quantum programming. I would like to explain to you the basics of this exhilarating feature because I am sure that it will be very important in the future, and I hope that you will become passionate about this topic.

I will try to make my explanation as simple as possible, although we are dealing with a complex subject. In order to better understand this section, you need to carefully read *Chapter 2*, *Symmetric Encryption Algorithms*, in particular, the sections that refer to operations in Boolean logic, and *Chapter 9*, *Quantum Cryptography*.

In the next three sections, we will talk about the following subjects:

- Classical information
- Quantum information
- Quantum gates and circuits

Let's talk now about classical information so we can compare it with quantum information.

Classical information

As we have already seen throughout this book, in particular when we talked about logical and Boolean gates in *Chapter 2*, classical information can be carried by bits: 0 and 1. We will focus here on analyzing what happens in a physical system (both classical and quantum) when it stores information. We will call this stored information X and we will call its state S.

We are analyzing classical information, and we assume that X can be only in one of a finite number of **classical states** at each moment. Later, we will analyze qubits and their quantum states. When we refer to a classical state, we denote the configuration of the system that can be described unambiguously without any uncertainty or error.

At the mathematical level, we simply define the set of classical states. We choose them in whatever way best describes the device that we are working with. So, we've given the name S to the finite set of classical states, and we will measure which state X will be in.

For example, it could be that X is a bit. In this case, the set of states contains two elements, and we write it mathematically as follows:

- If X is a bit, then its classical state S can be written as follows:

$$S = \{0, 1\}$$

- If X is a switch on a device in your home, such as a fan, maybe the classical states are *high*, *medium*, *low*, and *off*:

$$S = \{high, medium, low, and\ off\}$$

These are just examples of classical information carried by a classical system set in different states depending on what system we are working with, or which device we are using. These classical states are completely intuitive, and we have only to find a way to express them in mathematical terms. Mathematically, classical states are a finite set, and they must be non-empty as well. There must be at least one classical state in which the system can be in, for example, "*on*" or "*off*." There also have to be at least two classical states that the system can be in, in order to store information, i.e., bits: {1, 0}.

Take a look at the following figure, where the filled-in circle represents a bit in the zero state:

Figure 10.1: Bit where S=0

This figure shows a classical bit that can be in only two states. In this case, the filled-in circle denotes that the state of the bit is zero, while the empty circle denotes that the alternative state of the bit would have been one.

At any moment there may be *uncertainty* about the classical states of a system, so each possible classical state has a *probability* associated with it.

For example, if *X* is a bit, then perhaps it is in state 0 with a probability of 3/4 or in state 1 with a probability of 1/4. These are *probabilistic states* of X, and we can represent them mathematically as follows:

$$Pr(X = 0) = 3/4 \; and \; Pr(X = 1) = 1/4$$

There is a more succinct way to describe this probabilistic state with a **column vector**:

$$(3/4) <——— \; probability \; corresponding \; to \; [0]$$

$$(1/4) \; <——— \; probability \; corresponding \; to \; [1]$$

For example, if we flip a coin a number of times randomly, we will obtain roughly 50% heads and 50% tails. Keep in mind that randomness is very difficult or impossible to achieve in a classical system. We will see, however, that it is possible to obtain a completely random qubit generator using a quantum algorithm that we will implement later. However, here, supposing the coin is flipped randomly (because we are modeling a theoretical classical system), and supposing it is fair and not biased, we will have 1/2 probability of *state 0= heads* and 1/2 probability of *state 1=tails*.

We can represent the states of randomness as a line of junction between the two bits, 0 and 1, as shown in the following figure:

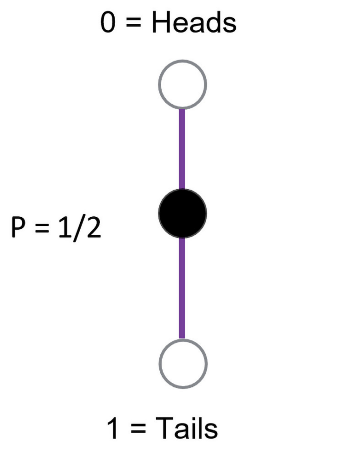


Figure 10.2: Probability of obtaining heads or tails in a random coin flip

This figure represents the probabilities of all the states of the coin before the flip. After the flip, the classical coin will assume a certain and deterministic value. We can expect to see **½** (half) of the time the probability is **Heads** and half of the time it is **Tails**.

This is just a representation of the probability of a bit (0, 1) in a classical state. As we can see, the classical state is very linear and predictable.

We can also do some manipulations to classical information and use Boolean logic (as we saw in *Chapter 2, Symmetric Encryption Algorithms*) to change the state of the bit or to make some operations on it.

To get familiar with this representation, which helps us to visualize the states (and will lead to the quantum representation), there is a logic gate that we have already seen, which is the *NOT* gate.

With the Boolean *NOT* operation, we can flip the classical bit from state 0 to state 1:

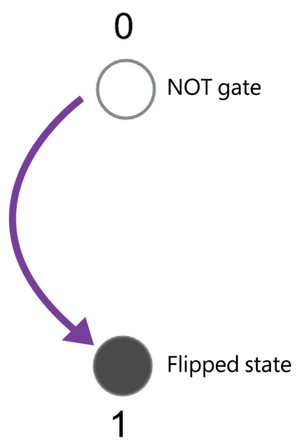


Figure 10.3: The NOT operation in classical bits

The figure represents flipping the state of a classical bit from 0 to 1 by using a *NOT* gate.

Let's see how all this operates in a quantum system and explore its peculiarities.

Quantum information, gates, and circuits

Just like the bit is the unitary information related to a classical computer, the qubit is the unitary base of information for a quantum computer.

The most important difference between classical information and quantum information is that a quantum system can be put in a **superposition** (as we have already seen in this chapter). Now, what I want to do is explain in a practical way what it means to put a system in superposition and how to achieve this by using and manipulating qubits with a quantum computer. In other words, I want to introduce you to the world of quantum programming.

In order to do that, I need to introduce the concepts of **quantum gates** and **quantum circuits**.

First of all, a quantum gate is similar to a classical Boolean gate, but we need to remember that in quantum computing, we are acting on a 3D sphere (a Bloch sphere), and no longer on a circle in 2D. That suggests a fundamental difference from classical computation: our particles (qubits) can be in more states than the only two states of the classical bit.

As we have already seen, quantum states are typically represented by **kets** in a notation known as **bra–ket**. If we recall the image of the Bloch sphere, we can say that the state of a qubit can be represented geometrically on any point on the surface of the sphere:

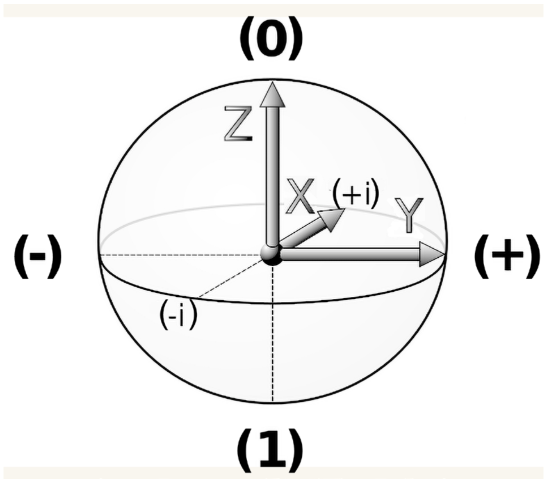


Figure 10.4: State of a qubit represented in a Bloch sphere

Let's explore quantum circuits, which are fundamental for programming quantum algorithms and quantum computing.

Pauli gates (X, Z)

A Pauli gate is composed of three different gates related to the rotation of a qubit with respect to the *X*, *Y*, and *Z* axes. If we analyze the sphere that represents the geometrical visualization of all the possible qubit states more deeply, we'll see that |0> and |1> lie on the *Z* axis, which is orthogonal with respect to the *X* and *Y* axes.

Suppose our qubit is in the initial state |0>, on the *Z* axis, as you can see in the next figure. Now, if we apply a quantum *NOT* operation (also called a **Pauli-X gate**), the state of the qubit will change similarly to the classical *NOT* gate. The state of the qubit, |0>, will flip to |1>. *Vice versa*, if the initial state of the qubit is |1>, applying a Pauli-X gate will flip it to |0>.

In the case of the initial state |0>, the qubit will be rotated 180 degrees clockwise with respect to the *X* axis:

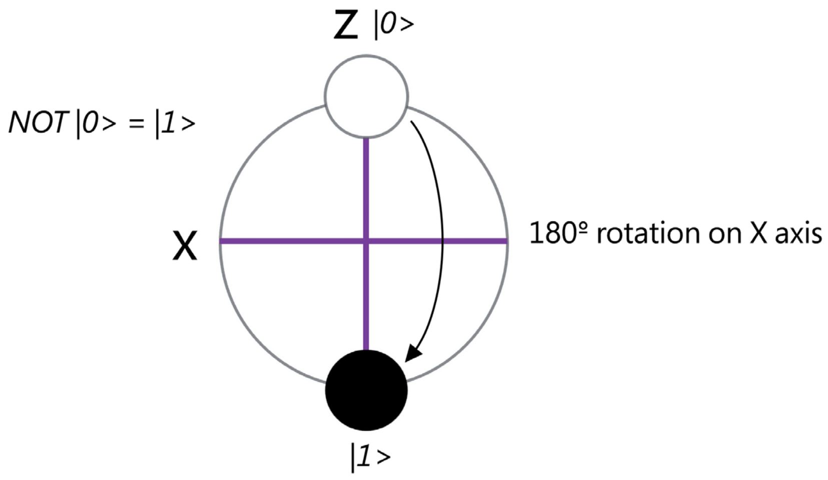


Figure 10.5: Pauli-X gate makes a rotation of 180 degrees on the X axis

As already mentioned, the Pauli-X gate is the quantum equivalent of the *NOT* gate in Boolean logic with respect to the standard basis |0>, |1>. With a rotation of the *Z* axis, this gate applies a 180-degree rotation of the *X* axis of the sphere.

The Pauli-X gate is identified by the following symbol:

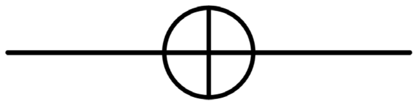


Figure 10.6: Pauli-X gate symbol

Mathematically, we can write the Pauli-X gate as a matrix like this:

$$X = NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The 3D representation of Pauli-X gate rotation is as follows:

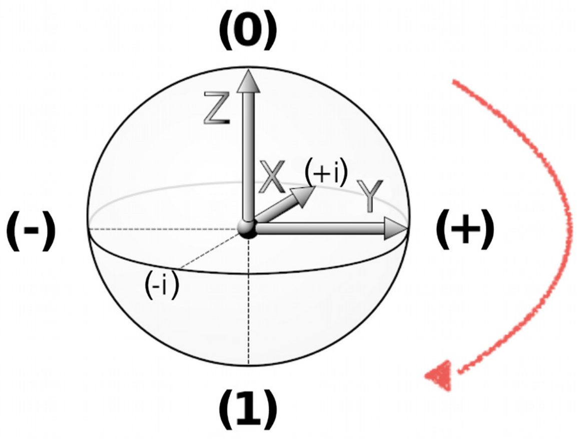


Figure 10.7: Pauli-X gate rotation on the X axis. It flips the state 0 with 1

If the rotation is made about the *Z* axis, then we have a **Pauli-Z gate**, identified by the following symbol:

Figure 10.8: Pauli-Z gate symbol

We can write a Pauli-Z gate as a matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Pauli-Z gates leave the basis state |0> unchanged and map |1> to |-1>. Due to this, Pauli-Z gates are sometimes called **phase-flips**. Pauli-Z gates are very important for implementing an oracle with Grover's algorithm. **Oracles** (which we will learn more about in the next section, when we talk about Grover's algorithm) are useful for increasing the probability of finding the item searched for.

We can also geometrically represent a Pauli-Z gate to better visualize this operation. In the next figure, you can see a 3D representation of the rotation on the *Z* axis and *X* axis, respectively. It changes the state from */->* to */+>*, while the states */0>* and */1>* remain unchanged:

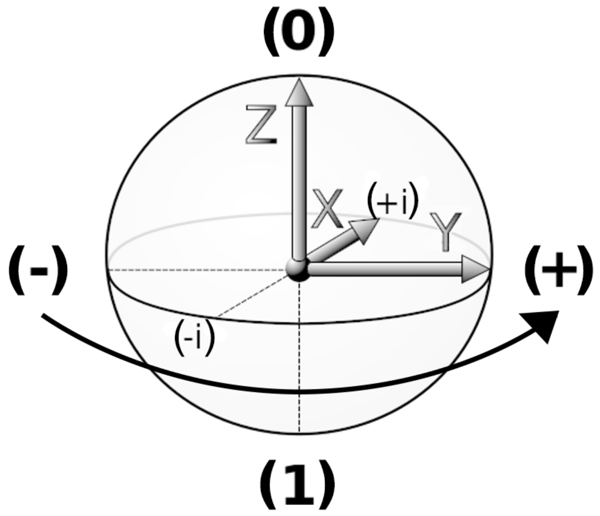


Figure 10.9: Pauli-Z gate rotating 180 degrees, from the state |-> into |+>

Identity gate

There is another quantum gate called the **identity gate**, which is opposite to the Pauli-X gate. It's a simple matrix that identifies the qubit in its initial state:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The identity gate is identified by the following symbol:

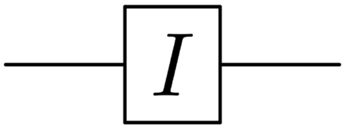


Figure 10.10: Identity gate symbol

Important note

As you may have noticed, the symbols are represented with a wire spanning from left to right. Wires carry information and connect the gates, which represent the operations applied to the information carried by the wires. You can think of a circuit just like an electric wire that connects various switches. In quantum computing, the circuits connect the logical quantum gates.

Hadamard gate

A very important quantum gate that is useful for implementing superposition and **Quantum Random Number Generators** (**QRNGs**) is the Hadamard gate.

The Hadamard gate imposes uniform superposition on our system. This gate is very important because often we need to scramble our initial state before starting the computation. The Hadamard gate does this for us in a predictable way.

The Hadamard gate provides a 90-degree rotation along the *X* axis. It flips our qubit from the */0>* state to the */+>* state. Take a look at the following figure:

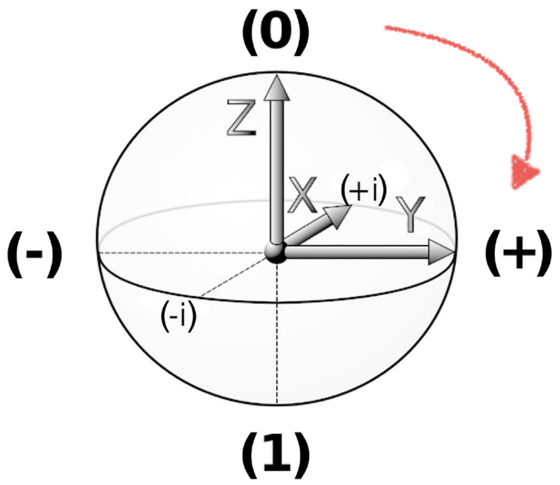


Figure 10.11: Hadamard gate on Bloch sphere

The Hadamard gate uses thefollowing symbol:

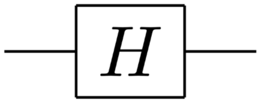


Figure 10.12: Hadamard gate symbol

The Hadamard gate is composed of all ones except for the last bit, which is –1. The gate is expressed with the following matrix:

$$H = 1/\sqrt{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Important note

Remember that $1/\sqrt{2}$ ensures that the sum of the probabilities of all possible outcomes adds up to 1.

With the Hadamard gate, as mentioned before, one potential application is the implementation of QRNGs. In fact, the Hadamard gate puts all the qubits on an equal superposition that is equivalent to a random possibility of assuming any state at the same time.

One of the problems in cryptography is the generation of random numbers. You might think that random numbers are easy to generate, but that is not true, especially in our physical, classical world. Think, for example, about tossing a coin. Usually, it is said that there is a 50% probability of heads and a 50% probability of tails. But that is not true. If the coin is tossed many times from the hand of the same person, it can be manipulated by that person, who could force the choice to alter the probability.

In quantum mechanics, things are totally different. By using the Hadamard gate, we can potentially program a quantum algorithm that can generate perfectly random numbers, although that is a topic of its own.

Now it's time to go deeper inside Grover's algorithm, analyzing its complex structure.

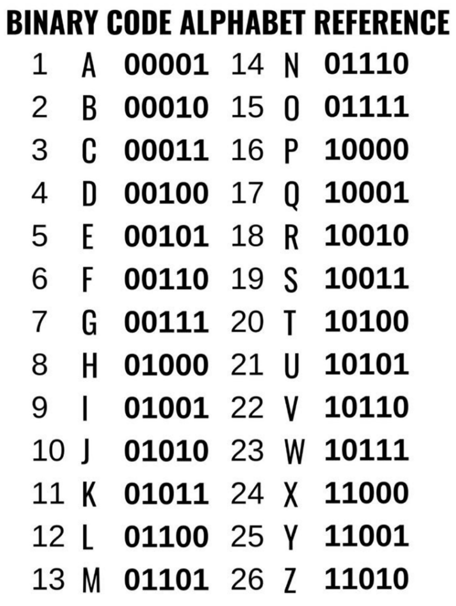
Deep dive into Grover’s algorithm

First of all, given the complexity of the topic and the relevance of this algorithm to the foundations of quantum algorithms and computation, I want to provide some notes about the notation and terms used in this chapter, and the approach that will help to formalize our mathematical formulas.

If we have a function *f* that maps binary strings of some length *N* to the binary code alphabet expressed in the following table, we refer to a **query model** problem searching inside an unstructured dataset. We have to find a **solution** to the function *f* (the input of our problem) inside an unstructured dataset. You might think of a solution of complicated equations that is able to crack an asymmetric or symmetric algorithm, or generically you can think of a combination that opens a lock of a strongbox. That is the scope of this algorithm in cryptography.

It is also useful to note that the solution might not even exist. This could happen when the name or the number we are seeking is not present in the dataset, so we simply note that there is no solution to the problem. However, to explain this, we will go through a search query that has a solution that is **unique**. This will better help me explain the **unique search problem**.

To begin with, let’s take a look at coding the alphabet in binary:



BINARY CODE ALPHABET REFERENCE

1	A	00001	14	N	01110
2	B	00010	15	O	01111
3	C	00011	16	P	10000
4	D	00100	17	Q	10001
5	E	00101	18	R	10010
6	F	00110	19	S	10011
7	G	00111	20	T	10100
8	H	01000	21	U	10101
9	I	01001	22	V	10110
10	J	01010	23	W	10111
11	K	01011	24	X	11000
12	L	01100	25	Y	11001
13	M	01101	26	Z	11010

Figure 10.13: The binary code alphabet

We start by saying that the total number of input strings in the function *f* is *N* expressed as the following power of 2:

$$N = 2^n$$

So, *N* is the square of the original length of the input string, *n*, to the function *f*.

Recalling the example proposed by Grover in his original paper, again, I am skeptical about this example because a search inside a register made of names is usually alphabetized; anyway, we will proceed in this direction, hoping it will not create confusion.

Suppose that we have a register of $N = 2^7$, therefore composed of 128 names related to phone numbers. We have the position of each name in the register on the left, then the names and the phone numbers for each name:

[1] John: 111 222 3333

[2] Phil: 222 333 4444

[3] Victor: 999 111 0000

.

.

.

[128] Elaine : 333 444 5555

You'll probably agree with me that if we are looking for the phone number of Elaine (the last position), the problem is the same as looking into an unsorted array of numbers (corresponding to each name in the register) randomly ordered (not alphabetical), like the following:

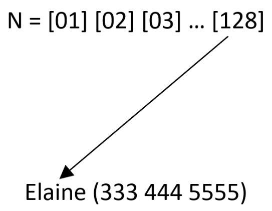


Figure 10.14: Random position of Elaine in the database

This is because we don't know where Elaine's number is positioned in the register.

The classical way to find the solution is to iterate through all *N* instances until we find the name Elaine in the last position [128]. In the worst case, we would scan all the numbers in the register from left to right until we arrive at [128] in the last position. So, if we divide the register into two parts, we have a 50% probability of finding Elaine in the first part and a 50% probability of finding Elaine in the second part:

$$50\% - - - \{[1, 2 \ldots 64]\} \quad \{[65, 66 \ldots 128]\} - - - 50\%$$

Remember that we know the instance target (output: Elaine) but we don't have any idea about the input's order or position in the register.

We can now apply Grover's algorithm to our search and see how the quantum system works and how much it can boost our search.

We divide Grover's algorithm into three main steps:

1. **Initialization** of the *n* qubits in an equal **superposition**. We will see in the next sections how to obtain the superposition and which gate to use in the quantum circuit. Now, we don't know anything about the function *f*, so we need to start with an equal probability to find the solution.
2. **Iteration**. Apply **Grover's operator, G,** *t* times (I will specify later how to determine the number of iterations *t*; for now, we assume that *t* is an integer and is arbitrarily selected). This is the heart of the algorithm. We will see in the next sections how to implement Grover's operator, which can also be called an oracle.
3. **Quantum measurement** and yield a candidate solution. If we choose *t* well, the probability of getting a solution will be high.

Let's analyze the algorithm's steps one by one, and you will see the fog start to dissolve.

Pseudocode for running Grover's algorithm

I want to show you some pseudocode for Grover's algorithm divided into the three main steps mentioned in the previous section:

1. Prepare a register *Q* of qubits all in the |0> state and **apply the Hadamard gate**, *H,* to put them in a state of superposition.

2. **Apply the Grover operation**, *G,* and iterate this operation *t* times, until you arrive at the closest chance of finding the best probability of success.
3. **Measure the new state** of the qubits and find the solution.

The next figure shows the entire circuit for Grover's algorithm. Don't worry if you don't yet fully understand it. After my explanation, you will be close to programming this algorithm by yourself using the **Qiskit** simulator tool from IBM.

Important note

In quantum circuits, the wires represent qubits and the gates represent both unitary operations and measurement. A circuit is a sequence of operations ordered from left to right.

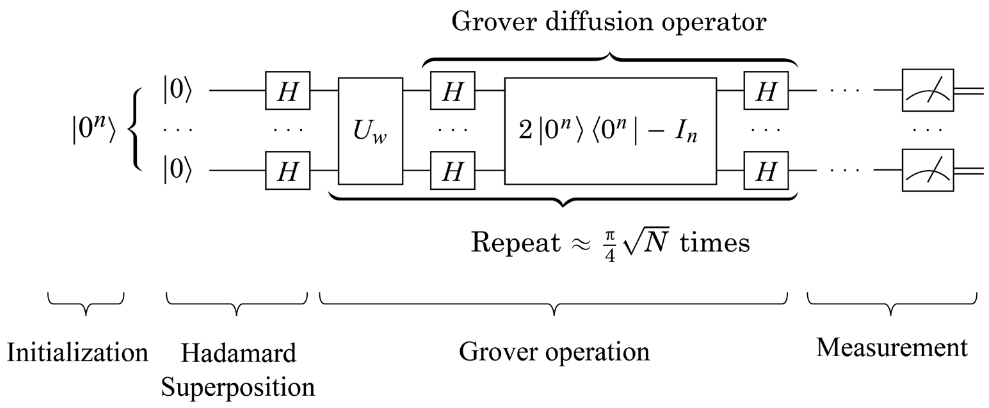


Figure 10.15: Circuit running Grover's algorithm with three stages of wires and gates: Hadamard superposition, Grover operation, and measurement

Let's explore the steps of this quantum search algorithm one by one.

Step 1: Allocate the register of qubits in superposition

The first operation we have to do is count the names in the register and allocate enough qubits to cover all the *N* instances (output names).

We generate the state of superposition using the Hadamard gate. In the zero state, |0>, it works as follows.

Since we assumed the number of instances is [128], we allocate 128 qubits in the register *N=128*, so we will need to allocate 128 qubits and put them all in superposition.

Let's talk a little bit more about the state of **superposition**.

Recall the mental experiment we discussed in *Chapter 9*, *Quantum Cryptography*, about the marbles that act both as particles and waves. More practically, the following explanation is the best I have found from MIT to describe what superposition is: *the ability to simultaneously be in multiple states.*

To put qubits into superposition, researchers manipulate them using precision lasers or microwave beams. In effect, we manipulate photon polarization and electron spin to obtain the superposition of all the particle's states.

This operation of putting a qubit in superposition is very similar to what we saw with Shor's algorithm in *Chapter 9*; here, we will analyze it more closely. If we have 128 qubits initialized with their states all zeroes, we call this number $N = 128$ and we mathematically represent it as follows:

$$N = 2^7 = 128 = |0{,}000 \ldots 0 >$$

Now, as we learned in the previous section, if we apply the Hadamard gate, we obtain the superposition.

So, just like in Shor's algorithm, we perform a transformation on each single bit of (n) inside the *bra-kets* until the n-bit, *in order to obtain the quantum state of superposition. So,* we get a sequence like this:

$$|u >= {}^{1}/_{\sqrt{2^n}} (|0000 \ldots 0 > +1000 \ldots 1 > + \cdots 1111 \ldots 1 >)$$

where the notation */u>* denotes the state of superposition of the register Q.

So, we will use the notation */u>* to denote the unitary state superposition.

As we don't know anything about the function *f*, we put all the qubits in the same state with equal probability using a Hadamard gate.

How and where can we use a Hadamard gate to perform the superposition?

As I introduced in the previous section, the Hadamard gate puts all the qubits in a state of uniform equal probability. I have plotted a four-qubit simulation using Qiskit in the next figure. As you can see on the right of the circuit, the probability is equal for all four qubits:

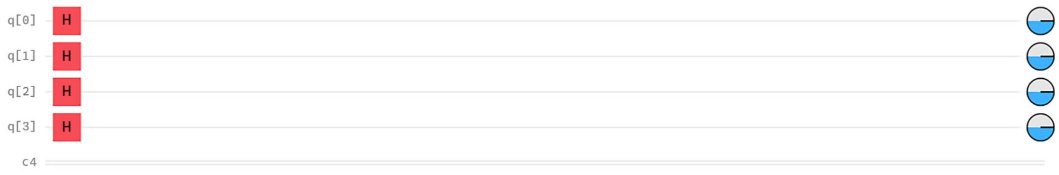


Figure 10.16: Hadamard gate in Qiskit

Important note

The four qubits used here are just an example. It should be 128 qubits, but that is too much to represent for a simple simulation.

Recalling the mathematical notation of the Hadamard gate, as we have previously seen, here is the matrix expression:

$$H = {}^{1}/_{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

As mentioned, the Hadamard gate puts all the qubits into superposition. In this case, I show how four qubits in uniform superposition have an equal probability of 6.25%. In fact, *6.25% *16 (all the possible states of the four qubits) = 100%*. That is represented in the following figure using the Qiskit simulator from IBM:

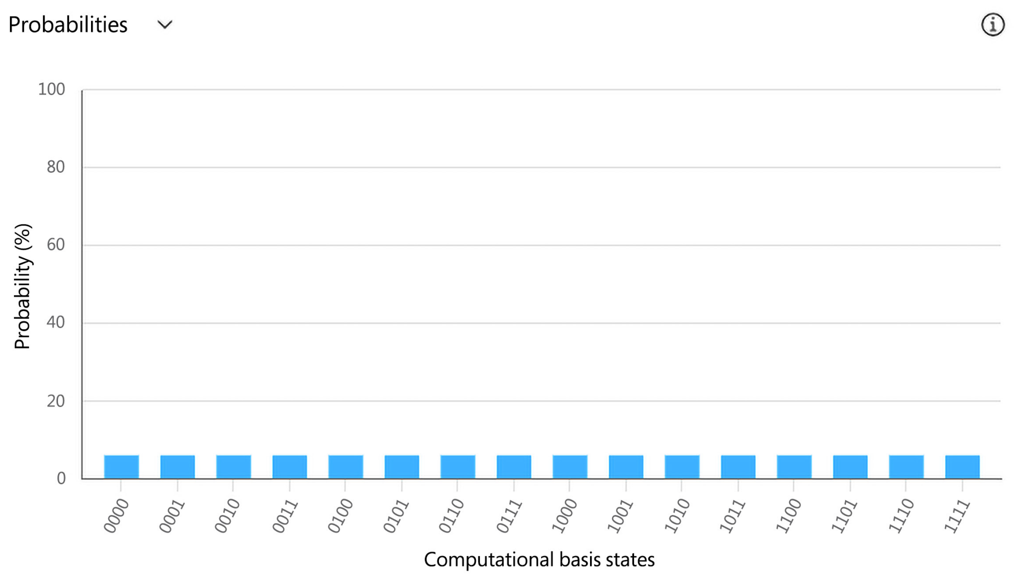


Figure 10.17: Plot showing all 16 states of the basis, each one with an equal probability of 6.25%

In the following figure, you can see the same condition of the qubits having equal superposition in a representation on the Bloch sphere (represented in Qiskit):

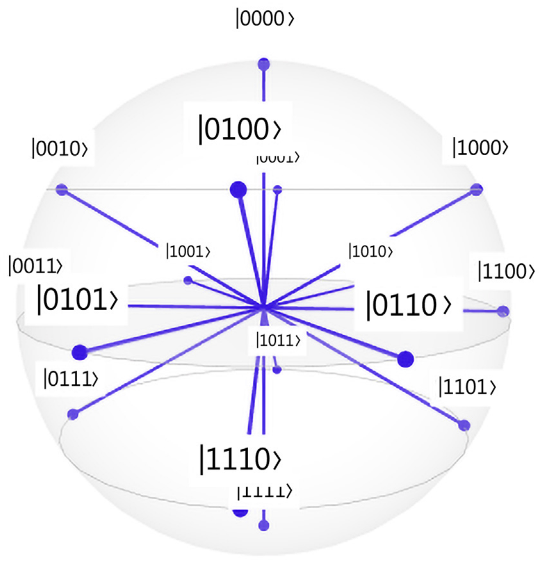


Figure 10.18: Visual representation on the Bloch sphere of all 16 states of the 4 qubits in superposition with equal probabilities

Now that we have prepared the register *Q* and put the quantum system in superposition, we will proceed with the next step: implementing the oracle.

Step 2: Iterations on Grover's operator: G

This is the hard part of the algorithm, and I understand it may be a struggle to go through. This is because we need to handle concepts and terms that are mostly counterintuitive or not quickly understandable.

After having put our 128 qubits into superposition, all the states of the qubits have an equal probability of being selected. So, we are in the same position as at the beginning. We still haven't advanced our search.

Returning to our example of a register of 128 names, in which we are searching for Elaine, remember that we know the output that is the solution to our query: Elaine. Now we need to *maximize the probability of finding the output*.

In order to maximize the probability of finding the output, we need to apply a special technique called **amplitude amplification**. In geometrical terms, this is an iteration that allows us to gain the closest position possible to maximize the probability of finding the target quantum final state (the solution). So, we need to calculate the element *t*, the number of times we need to iterate *G***,** the Grover operator, which I am going to explain in the next section. But before that, we need to dive into some particular components of this algorithm: the query model of computation and oracle.

The query model of computation

Now we are approaching a problem in quantum computation that is related to the use of a quantum computer. The question that comes naturally is: why should we use a quantum computer instead of a classical computer?

The answer could be that, in some cases, quantum computers are faster than classical computers at solving some specific problems. In the case of quantum search, we will see that a quantum computer is able to use Grover's algorithm exponentially faster. In particular, we are mainly focused on the time required for computation.

The model of computation we are going to explore is related to a **query computation model.** This model is different from the classical way we imagine computation, where we have *input* → *output*.

The model we looked at is made up of queries received as input and an output given by the answer to the problem, expressed by true or false, yes or no… more similar to a Boolean circuit than a computation that gives us the result of a precise mathematical operation. We call the "secret" process of computation an **oracle**, and we will see now how it works.

Take a look at the following figure, which depicts the computational model:

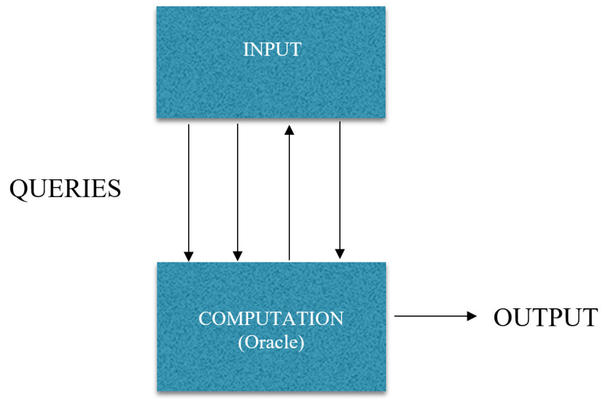


Figure 10.19: Computation model referred to as a query model

An oracle refers to a model in which for any given input, the oracle produces a single output as an answer: *yes–no* or *true–false*.

It's pretty strange that such a method can exist, because I'm sure you will agree with me that it is somewhat weird and counterintuitive. Indeed, if we add an input (as a query) inside a mechanism of computation such as an oracle and this mechanism gives an output, you will probably wonder how the oracle is able to do that.

There was a priestess called Pythia in Ancient Greece who was known as the **Oracle of Delphi**. If you asked this oracle something, she responded with an answer, but she didn't reveal all she knew about the problem. Similarly, a black box is a system that processes an input and gives an output without revealing anything about the process of determining the output.

In a mental experiment that helps us to reduce the complexity and simplify the way to figure out what an oracle is, we can think about the following problem.

We associate any name from our register Q (John, Phil, Elaine, and so on) with a geometrical figure: a circle, hexagon, triangle, and so on. The problem now is no longer finding the correct name but identifying the correct figure associated with it. For example, if we associate Elaine = right-angled triangle, John = circle, and Phil = square, we need to find the right figure inside a set of different figures, such as circle, rectangle, square, and hexagon.

One important condition is that the oracle "knows" the solution (just like we know the output is Elaine) and makes a filter that allows only the right-angled triangle to enter inside. So, following the proposed query, "find Elaine," the oracle filters for the right-angled triangle and finds Elaine. Take a look at the next diagram depicting the oracle filter and see if that helps you understand what an oracle is:

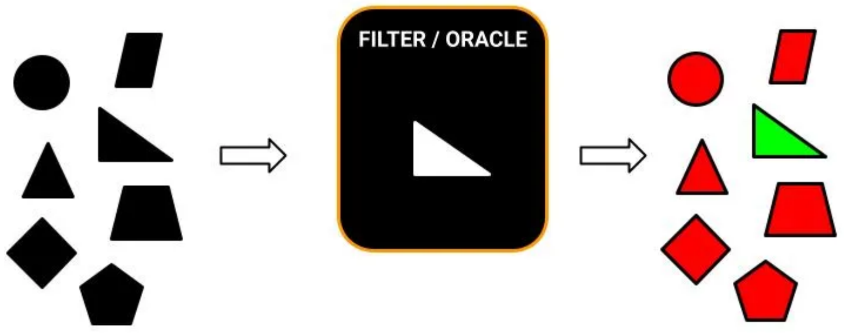


Figure 10.20: Oracle filtering for a right-angled triangle

However, there is still an issue with this model. The oracle can find the solution, but it still hasn't improved the speed of our search. Indeed, if Elaine is in the last position and so is the right-angled triangle, the cost of the computation will be the same as the classical computation. However, the oracle is useful for implementing a quantum algorithm because we will see that it is able to configure our system correctly with respect to our goal.

To be more specific and mathematically correct, the input takes the form of a function f, and we can evaluate this function f. But otherwise, we don't have any knowledge or understanding of how it works.

Now we will explore an example of a query problem: the **unique search problem**, and you will see that it is much simpler than what we have just seen.

The unique search problem and the amplitude amplification probability

Just like we saw in the Boolean circuit, the *OR* function gives some output to some input introduced in the function. Thankfully, this problem tends to be relatively more intuitive to understand.

In this case, we have a function f with n qubits as input.

The output is as follows:

- 1: If there is a string of qubits inside the function *f* for which *f(x)= 1*
- 0: If there isn't such a string of qubits

We now approach the unique search problem, which is related to the *OR* problem but is not exactly the same.

Step 1: The promise of the inputs

Here, we have a **promise** on the inputs; inputs that don't satisfy the promise are considered "don't care" inputs. In other words, we are seeking a unique string (*z*) that satisfies the following conditions for a unique search:

- **Input**: *f: ∑*n→*∑*, where *∑*n denotes the binary alphabet of length *n* and the function *f* maps binary strings of length *n* to the binary alphabet ∑
- **Promise**: There is exactly one string *z* that belongs to *∑*n for which *f(z)= 1*, with *f(x) = 0* for all strings *x ≠ z*
- **Output**: The unique string *z*

Now, based on this promise, we make a partition of the set of all the bit strings into two sets – *A0* and *A1*:

- *A1* contains the solution to our search problem
- *A0* contains all the non-solutions

We are particularly interested in two quantum states:

- */A0>*: **Uniform superposition** over all non-solutions
- */A1>*: **Uniform superposition** over the unique solution

As we have seen, in quantum computing, when we apply a quantum gate, usually we change the state of the qubit. In other words, we manipulate the spin of the particles or the polarization of the photons involved in this process in order to perform a reflection or a rotation of the angle in the Bloch sphere. Since this change of state is achieved by applying quantum gates such as a Hadamard or Pauli-Z gate, we will take into particular consideration these effects of changing state to show how to determine the amplitude amplification of the probability of getting our solution. In particular, we will see how the angle will rotate when we perform any step of Grover's algorithm and how this angle rotates when we perform the iterations.

After having explained all these issues, we can go on to explore the next step of the algorithm: the *G* operation.

Step 2: Grover's operation G

In this section, I will show the geometrical action of *G* iterated *t* times to get the solution. I will be focused mainly on the mathematics related to finding the angle θ (theta) and its **reflections/ rotations** that enable the **amplification of the probability** of finding the solution.

First of all, let's see the result in a geometrical representation of the effects of the two sub-steps of *G*:

- **Sub-step 1**: The oracle
- **Sub-step 2**: The probability amplification

The effects of *G*, as I have already explained in mathematical terms, are as follows:

- To map the output (oracle), as shown in *Figure 10.20*
- To amplify the probability in order to obtain the solution (diffusion), as shown in *Figure 10.21* and *Figure 10.22*

In the next figure, I show the effect of applying a minus sign to the target state */11>* expressed in an example of four possible states (oracle):

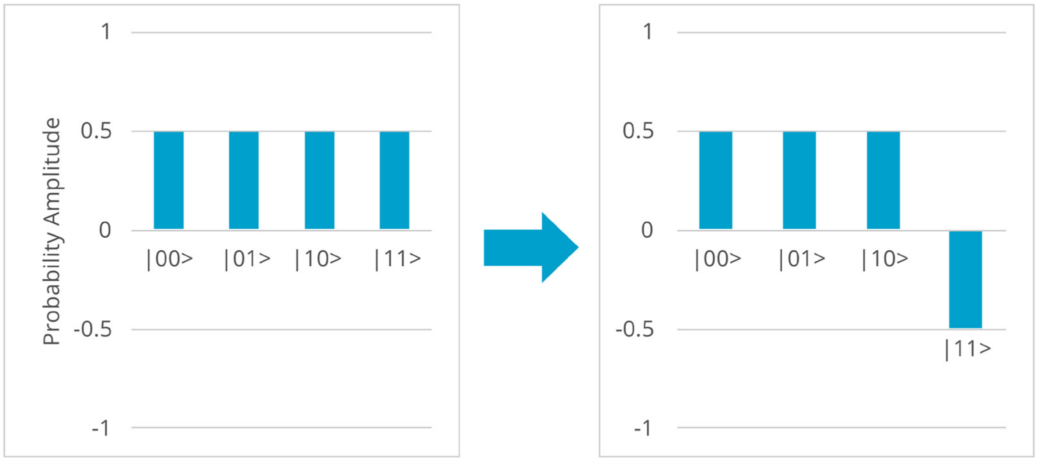


Figure 10.21: Probability amplitude 1

In the next figure, we see the effect of the amplitude amplification, which, over the four states of */00>*, */01>*, */10>*, and */11>*, brings the probability of obtaining the solution */11>* to 1, meaning it has a probability of 100%:

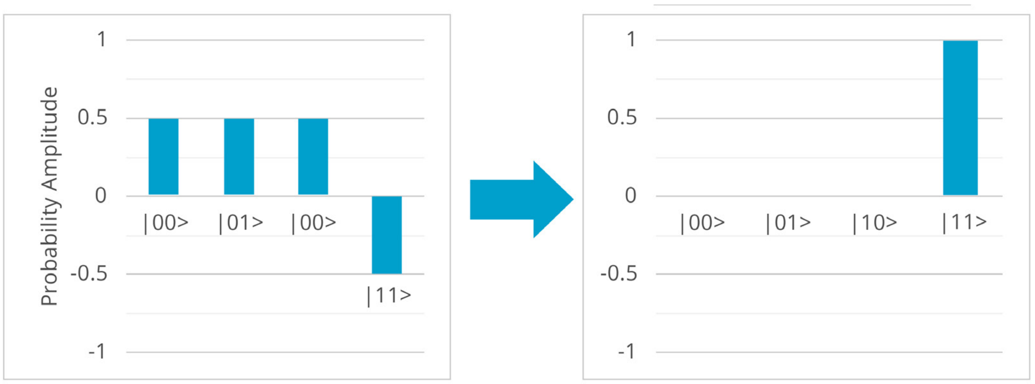


Figure 10.22: Probability amplitude 2

Now, we need to identify the angle θ and its expression.

The action of *G* is a rotation about a given angle theta as a matrix. The expression that denotes the theta angle is as follows:

$$\theta = \sin^{-1}\left(\sqrt{|A1|}/N\right)$$

And, because the rotations are effectively made in a two-dimensional subspace, I will show geometrically, one by one, the effect of each *G* iteration in order to seek the winning string *z*, also denoted as Elaine's phone number.

Remember that with the notation */u>*, we identify the state of superposition of our register *Q*. So, after the initialization step, and before any iteration of *G*, we will have the following condition on our qubits:

$$|u> = \cos(\theta\ |A0>) + \sin(\theta\ |A1>)$$

That is the state we start with.

At the first iteration *G*, we rotate by 2θ, so the total angle will be 3θ:

$$G\ |u> = \cos(3\theta\ |A0>) + \sin(3\theta\ |A1>)$$

At the second iteration *G2 |u>*, we apply another rotation of 2θ, so the total angle will be as follows:

$$G2\ |u>= \ \cos(5\theta\ |A0>) + \sin(5\theta\ |A1>)$$

Then, after *t* iterations, the angle will be:

$$Gt\ |u>= \ \cos\big((2t\theta + 1)|A0>\big) + \sin\big((2t\theta + 1)|A1>\big)$$

Basically, we apply a rotation of 2θ for any iteration of *G*.

In geometric terms, the following diagram shows a reflection of angle θ with respect to its initial state:

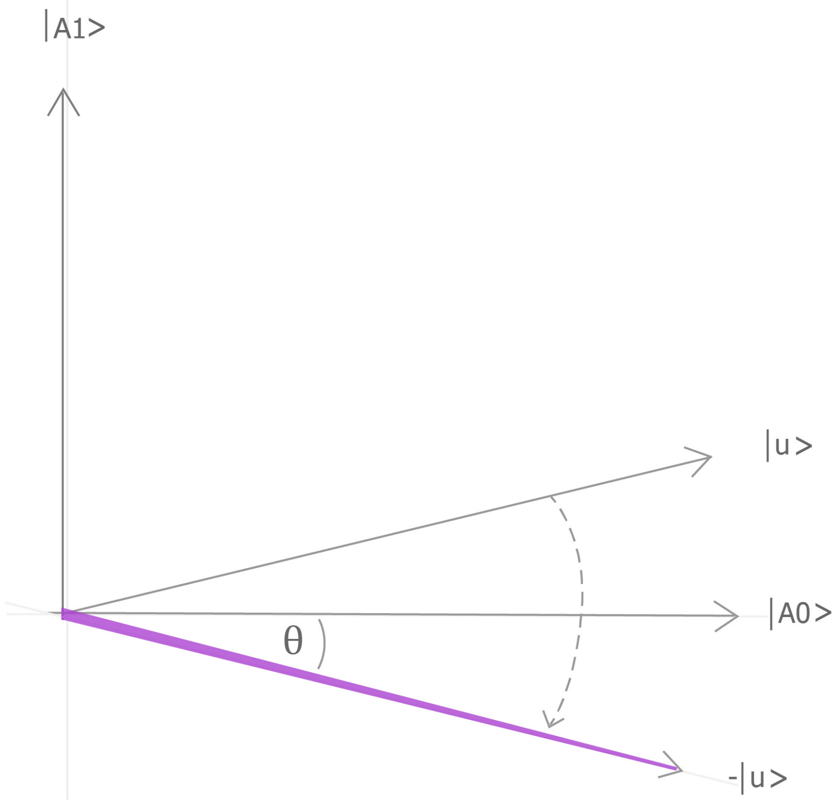


Figure 10.23: Geographic representation of the reflection of the θ angle

So, because the coefficient of *|A1>* is *sin(2t*θ*+1)*, we will see a solution *|A1>* with a probability *p* of:

$$p = \sin^2\big((2t + 1)\theta\big)$$

We should make this probability large, ideally as close to 1 as we can. So, as we are rotating the state vector by iterating *G*, we will think about */A1>* as the target state. Moreover, we have to keep in mind that we also want to minimize *t* because that is the number of queries we will need to make to reach the target state.

You may think of the number of queries *t* as similar to how many geometrical figures have to pass into the filter/oracle in *Figure 10.20* to catch the solution *z*.

So, if we want to make this probability close to 1 and minimize *t*, it makes sense to make the angle of our state vector close to this:

$$\left((2t+1)\theta\right) = \pi/2 <---> t = \pi/4\theta - 1/2$$

$\pi/2$ is the smallest angle that maximizes the probability p, and offers the best probability of reaching our goal of getting the target */A1>*.

I should say that the angle doesn't have to be π/2 exactly because *t* has to be an integer and π/2 is not an integer. So, approximately choosing an integer close to *t = π/4*θ *- 1/2* makes sense.

But of course, *t* must be a non-negative integer, so we ideally will choose *t* with the following value:

$$t = \pi/4\theta$$

This is a positive integer that will give fewer iterations of *G* to reach our goal.

But there is another important consideration that we need to make. This magic number of iterations *t = π/4*θ is valid only for unique search. In other words, we know that there is a solution, and this is only one single solution.

We have discovered now why we refer to the unique search problem in this explanation.

For the unique search, we have *s = |A1> = 1* and therefore we will need to consider the following equations for the theta angle:

$$\theta = \sin^{-1}\left(\sqrt{|A1>/N}\right) \approx \sqrt{1/N}$$

As the theta angle is equal to the inverse sin of the square root of */A1>/N*, as expressed in the above equation of theta, for smaller and smaller angles, the sin function looks more and more like the identity gate.

Substituting $\theta = \sqrt{1/N}$ into the equation of *t = π/4*θ gives us the following:

$$t \approx \pi/4 * \sqrt{N}$$

So, now we have a defined number related to *N* to approximately calculate *t*. And that is good because, as we said, the algorithm performs more or less as *O(√N)*.

Let's see with an example by numbers what happens in geometric terms when we iterate the theta angle in order to get the solution.

We have *N=128* strings corresponding to 128 names in an unstructured form:

$$N = 2^7 = 128$$

To see how the algorithm works, we need only to figure out the angle, theta, which we can calculate with the following mathematical expression:

$$\theta = \sqrt{1/N} = \sqrt{1/128} = 0.0885 \ldots$$

If we plug this number inside *t*, we get the following:

$$t \approx \pi/(4/\theta) \approx 8$$

So, we have eight queries to make in order to get the solution. Let's go find the solution!

The following figure shows the initial state of the superposition of */u>*. Remember that */A1>* is our target solution and */A0>* represents the uniform superposition of all the strings.

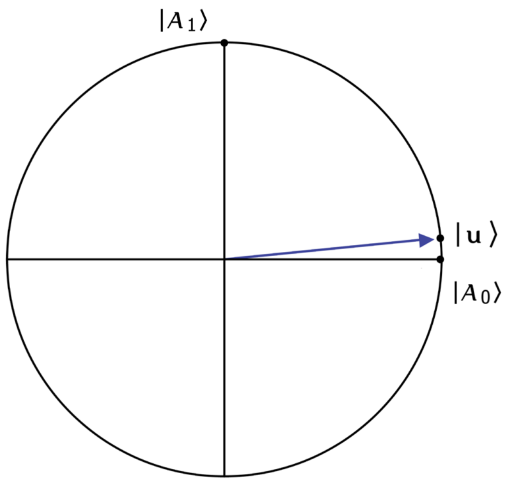


Figure 10.24: State of superposition of |u>

/A0> and */u* are close, as you can see, because */A0>* is almost in the state of superposition over all the strings. Only one is missing, and this is the unique solution.

By applying *G* repeatedly, in 8 iterations, we continually rotate towards */A1>*. Finally, at the eighth iteration, we reach the closest state to the solution */A1>*. These 8 iterations on 128 strings are the best optimization that we reach using Grover's algorithm:

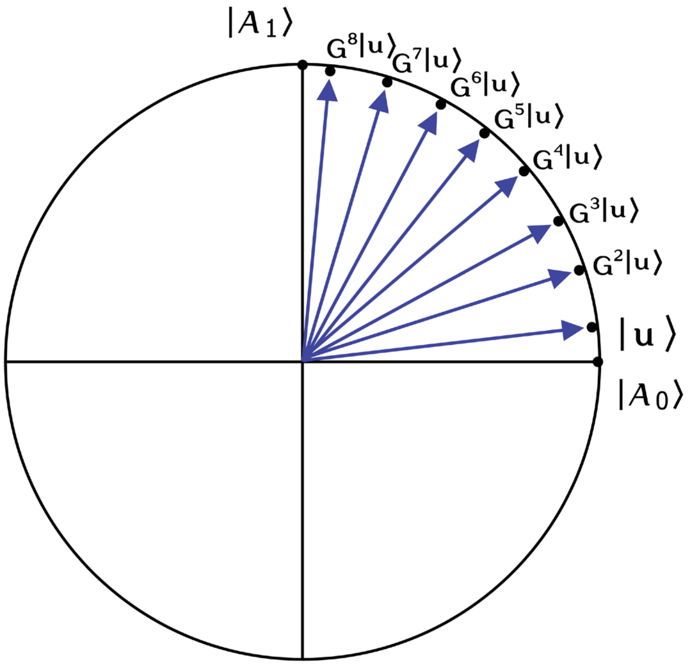


Figure 10.25: The eight iterations of G to reach the solution

At the eighth iteration, we reach the closest point to the string solution */A1>*.

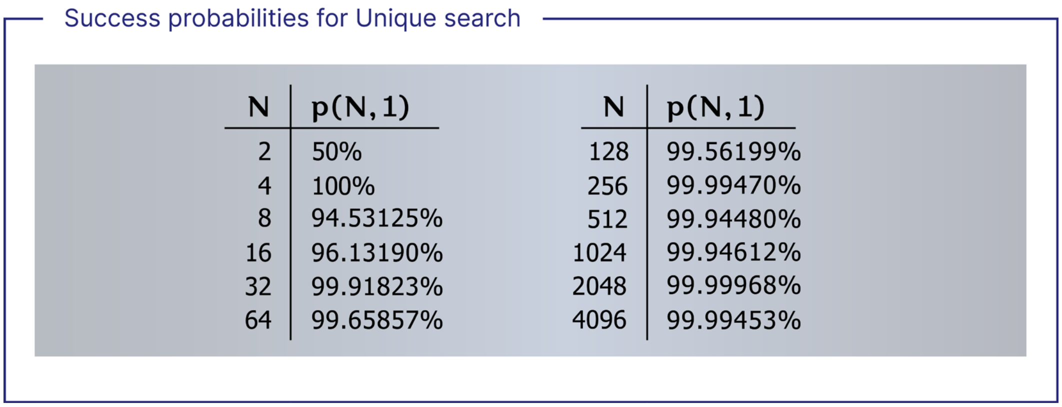
Let's now see what happens when we make the quantum measurement.

Step 3: Quantum measurement

If we stop and measure at this point, we get the solution */A1>*. In fact, in this case, we have a 99.5% chance of getting the solution. If we continue to iterate, we will get the opposite effect and we will move away from the solution.

In the following table, you can see the probability for small powers of 2. Remember that $128=2^7$.

We see that in the case of *N=2*, the probability is 50%, so it doesn't work so well, but for *N=4* the probability is 100%. In this case, with only 1 query, we get the solution.



Success probabilities for Unique search

N	p(N,1)	N	p(N,1)
2	50%	128	99.56199%
4	100%	256	99.99470%
8	94.53125%	512	99.94480%
16	96.13190%	1024	99.94612%
32	99.91823%	2048	99.99968%
64	99.65857%	4096	99.99453%

Figure 10.30: Probability of different N values

As you can see in the table, for a number of strings between 1000 and 4000, the quantum measurement gives a probability very close to 100% (>99.9%).

Summary

In this chapter, we have analyzed quantum computing and searching algorithms for quantum cryptography. After introducing the basic principles of quantum programming and quantum gates, we discovered how to build Grover's algorithm for quantum search.

However, Grover's algorithm hasn't yet improved on the process of exponentially searching an unstructured database; it is only a great starting point for creating new and more efficient quantum algorithms. As we have seen in this chapter, this model of searching is not yet feasible, given the high number of qubits required to obtain the target solution. When implemented, the possible future applications of quantum search in cryptography will be potentially devastating for classical cryptographic algorithms. Indeed, the problem of searching in an unstructured database is the same as brute-forcing a dataset of keys inside a cryptography symmetric algorithm. So, in the future, if the quantum wave rises, many classical cryptographic algorithms will probably be swept away.

Now that we have reached the end of this book, I offer my deep congratulations to you for having followed my thoughts, some tough demonstrations, and explanations of algorithms. I hope I have made cryptography a little clearer. I suggest you keep reading and learning! In particular, I invite you to visit the QuantumLab website that I am involved in (`www.quantumlab.science`) and connect with our community of quantum cryptography enthusiasts

Other Books You May Enjoy

Index

A

Advanced Encryption Standard (AES) 260
AES256 337
AES Rijndael 64
  attacks and vulnerabilities 69-74
  description 64-69
algorithm 4
amplification of probability 364
amplitude amplification 360
AND logical conjunction 111
appendix signature method 210
ARPA 79
ARPANET 79
ASCII code 7-9
asymmetric algorithm
  protecting, from spying method 205, 206
asymmetric encryption 6, 77, 78, 320
attack analysis 318-320
authentication 124, 125
Authority for Communications Security (COMSEC) 78

B

Beale ciphers 20-26
big O notation 342
binary numbers 7-9
birthday attack 87
Bitcoin 109
bit encryption 49, 50
bit expansion 46-48
bit initialization 46
bit padding 94
blind payments 292
blind signatures 133, 134
  protocol 136, 137
  with RSA 134-136
blockchain 174-178
block cipher 44
block-reply 71
book cipher 23
Boolean logic 281, 282
  notations and operations 38-43
Boolean satisfiability problem (B-SAT) 38
bra-ket 347

C

Caesar cipher 16-19
cipher 4
cipher block chaining (CBC) 64
cipher feedback block (CFB) 64

ciphertext-only attack 28

classical states 343

code-based cryptography 337

collapse of the wave function 307

compression function 116

computational brute-force cracking
example 290-292

confusion 51

consensus problem 177

Copenhagen interpretation 304

counter (CTR) mode 64

CovidFree 292

cryptographic algorithms 14
Beale ciphers 20-26
Caesar cipher 16-19
Rosetta Stone 14-16
ROT13 19, 20
security and computation 29-32
Vernam cipher 26-29

cryptography 4-7, 300
ASCII code 7-9
binary numbers 7-9
Fermat's Last Theorem 9-14
modular mathematics 9-14
notations 7-9
prime numbers 9-14

Crypto Search Engine (CSE) 64, 166, 263-266, 282-284
applications 292-294
computational analysis 288, 289
computational brute-force cracking, example 290-292
innovation 284-288

cryptovirology 198

Cybpher
test and performance 229-231
using, in Lightweight Encryption Algorithms (LEA) 227-229

Cybpher algorithm 225, 226

cyclic group 243

D

Decentralized Autonomous Organization (DAO) 216

decryption 5

Defense Advanced Research Projects Agency (DARPA) 79

DES algorithms 43-51
analysis 57-60
bit encryption 49, 50
Bit expansion 46-48
Bit initialization 46
key generation 48, 49
violation 60, 61

DES cracker 60

DESX 63

Diffie-Hellman (D-H) algorithm 80-85, 157, 158
analyzing 85, 86
attacks and cryptanalysis on 86-88
base point 243
discrete logarithm 82-84
generator point 243
implementing, on elliptic curves 242-244
numerical example 245-248

digital signature 124, 125, 320
with ElGamal algorithm 130-133

Dirac notation 307

direct signature method 208, 209

discrete logarithm 30, 82-84

D-Waves 323

E

Electronic Code Book (ECB) 47

Electronic Frontier Foundation (EFF) 60

electronic health records (EHRs) 292

ElGamal algorithm 100-103
- using, in digital signatures 130-133

Elliptic Curve Cryptography (ECC) 233, 234
- future considerations 261

Elliptic Curve Digital Signature Algorithm (ECDSA) 234, 248

elliptic curves 234
- attacks on ECDSA and security 258
- D-H algorithm, implementing 242-248
- operations 234-239
- overview 234
- private key, [d], recovering 259, 260
- random key, [k], discovering 258
- scalar multiplication 239-242

elliptic curve secp256k1 248-250
- digital signature, performing 251, 252
- digital signature, verifying 252
- keys, generating 251
- numerical exercise, on digital signature 252-258

encrypted data storage 292

encryption 5
- asymmetric encryption 6
- symmetric encryption 6

entanglement on demand 309

European Commission for Cybersecurity (ENISA) 336

exclusive OR (XOR) 27, 38

F

Feistel cipher 44

Feistel system
- structure 50

Fermat's Last Theorem 9, 179-183

Fermat's Little Theorem 13

field-programmable gate array (FPGA) 74

file synchronization and sharing (FSS) 292

first add round key (F-ARK) 65
- layers 65

Fourier transform (FT) 330

G

GNU Privacy Guard (GnuPG) 103

greatest common divisor (GCD) 335

Great Internet Mersenne Prime Search (GIMPS) 12

Grover's algorithm 339
- amplitude amplification probability 362
- binary code alphabet 354, 355
- example 291
- exploring 353
- initialization 355
- iteration 355
- iterations on G 360
- overview 340-342
- pseudocode 355
- quantum measurement 355
- qubit, allocating in superposition 356-360
- query computation model 360-362
- unique search problem 362
- unstructured search, issues 340

Grover's operator (G) 355

H

Hadamard gate 351, 352

harmonics 330

hash algorithms
  overview 109, 110
  SHA-1 algorithm 115-119
  SHA-1 notes 119, 120

hash functions 106-109, 281, 282, 337
  logic and notations, for implementing 110-115

hash of M 106

homomorphic encryption 42
  analysis 270, 271
  implications 270, 271
  Math and logic, behind search engines 271-276

homomorphic search 292

homomorphism 264-266
  in RSA 266-269

Huffman code 278-281

hypothesis 326

I

identity gate 350

imaginary experiment, Q-Mechanics elements 302
  entanglement 307-309
  indeterminacy principle 304-306
  spin 306, 307
  superposition 302-304

indeterminacy principle 304-306

Initial Coin Offering (ICO) 176

Initial Public Offering (IPO) 176

Institute of Electrical and Electronics Engineers (IEEE) 257

interactive ZKP 151, 152
  challenge, for disruptive attack on 155, 156
  demonstration 153, 154
  demonstration of 152

isogeny-based cryptography 337

K

Kerckhoffs' principle 29

kets 347

key expansion (KE) 65

key generation, in DES 51-54
  decryption 56, 57
  encryption 54-56

key initialization 185

L

lattice-based cryptography 337

leaves 276

light experiment 301

light traps 311

Lightweight Encryption Algorithms (LEA) 224
  with Cybpher 227-229

M

Man-in-the-Middle (MitM) attack 86, 87, 124

many-world interpretation 304

mathematical demonstration 211-214

MB09 algorithm 174-178, 183-191

MBXI algorithm 191-194
  numerical example 194-196
  RSA attack 196-198

MBXI digital signatures 206, 207
  appendix signature method 210

direct signature method 208, 209
mathematical demonstration 211-214

MBXX protocol 179, 214-221
functions 221-223
parameters 218
problem solving 223

MD5 Generator 110

measurement 305

modular mathematics 8, 9

M-Pesa 178

N

NAND logical operator 42

National Bureau of Standards (NBS) 43

National Institute of Standards and Technology (NIST) 43

National Security Agency (NSA) 43, 78, 109, 319
pioneers 79, 80

Netscape Communicator 20

nondeterministic polynomial (NP) 38

non-interactive zero-knowledge proofs 144-148, 161
attack, demonstrating on RSA ZKP 148-151
demonstration 148

notations 7-9

not equal 10

NP-complete 38

O

observation 305

observer 305

one-round ZKP 156, 157
mathematical usage 157

one-round ZKP, mathematical usage
numerical example 157, 158
one-round protocol 159, 160

one-time pad (OTP) 28

one-way function 85

Oracle of Delphi 361

OR logical disjunction 111

P

padding 94

pattern recognition 340

Pauli gate 347

Pauli-X gate 347, 348

Pauli-Z gate 349

perfect cipher 26

periods 330

phase-flips 349

polarization of the photon 310

polaroid 310

polynomial time (P) 38

post-Q-Cryptography 336, 337

Pre-Shared Key (PSK) 166

Pretty Good Privacy (PGP) 99, 100

prime numbers 9

probability 306

Proof of Concept (PoC) 191

proof of knowledge 142

Proof of Work (PoW) 109

pseudocode
for running Grover's algorithm 355

public parameters 185, 186

Q

Q-Cryptography 300, 326
  origin 310-312
Q-Mechanics 300
quantum bit (qubit) 307
quantum circuits
  Hadamard gate 351, 352
  identity gate 350
  Pauli gate 347
  Pauli-X gate 347, 349
  Pauli-Z gate 349
quantum computers 323
quantum computing 314, 323-325
quantum entanglement 307
quantum Fourier transform (QFT) 330-334
quantum key distribution (QKD) 319, 320
quantum key exchange 312
quantum message system transmission 321
quantum money 310
quantum programming elements
  circuits 342
  classical information 343-345
  quantum circuits 347
  quantum gates 346
  quantum information 342, 346
Quantum Random Number Generators (QRGNs) 351
quantum search algorithm
  qubit, allocating in superposition 356-360
quantum transmission key algorithm
  photons, transmitting 315
  quantum channel, initializing 315
  shared key, determining 316-318
Quantum Transmission Message (QTM) 294
qubit 313, 314
query computation model 360-362
query model problem 353

R

replay attack 171
Righetti Computing 323
RIPEMD-160 109
Rivest, Shamir, and Aldemar (RSA) 88-91, 321
  analyzing 91
  application, to verify international treaties 94-96
  conventional attacks 92-94
  homomorphism 266-269
  unconventional attacks 96-98
  using, for blind signatures 134-136
Rosetta Stone 14-16
ROT13 19, 20, 37
RSA algorithm
  protecting, from spying method 205, 206
RSA digital signatures 125-128
  usage, significance 129
RSA ZKP
  attack, demonstrating on 148-151

S

salt 72
scalar multiplication 239-242
Search for Extraterrestrial Intelligence (SETI)
  URL 15
secp256k1 233

Secure Hash Algorithm 1 (SHA-1) 115-119
notes 119, 120
Secure Hash Algorithm (SHA) 109
security and computation 29-32
self-reverse decryption, on RSA 198-200
key generation 201
private key 201-204
public key 201-204
SHA-1 example
one block encoded 121-123
Shor's algorithm 326, 335, 336
factorizing (n) 335
hypothesis 326
quantum measurement 329
qubits, initializing 327
random number, (a), selecting 328, 329
right candidate, (r), selecting 330
thesis 327
side-channel attacks 73
list 73, 74
spin 306, 307
spooky action at a distance 308
spooky moon math 160
spying method
used, for protecting asymmetric algorithm 205, 206
used, for protecting RSA algorithm 205, 206
Standards for Efficient Cryptography Group (SECG) 248
substitution (S) 44
superposition 302-304, 342, 346, 355, 357
symmetric encryption 6

T

thesis 327
time domain 330
tree graph 276-278
Triple DES 61, 62

U

unconventional attacks 198-200
unequal 10
unidirectional function 107
uniform superposition 363
unique search problem 353
example 362
Grover's operation G 364-369
Groverҳ operation G 365
promise of inputs 363
quantum measurement 369, 370

V

Vernam 321
Vernam cipher 26, 37
Virtual Machine (VM) 166

W

waves 303
Wolfram Mathematica 9

X

XOR bitwise sum 111
XOR logic operator 43

Z

zero-knowledge proof (ZKProof) 224

Zero-Knowledge Protocols (ZKPs) 142
- digital cave 142-144

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARKs) 161

ZK13 protocol 166-170
- demonstrating 170, 171
- possible attacks 171

ZK-proof of authentication 166

zk-STARKs protocol 160-162
- attack, demonstrating on 163-165
- usage 162, 163