Pro Cryptography and Cryptanalysis with C++20

Creating and Programming Advanced Algorithms

—

Marius Iulian Mihailescu
Stefania Loredana Nita

Apress®

Pro Cryptography and Cryptanalysis with C++20

Creating and Programming Advanced Algorithms

Marius Iulian Mihailescu
Stefania Loredana Nita

Apress®

Pro Cryptography and Cryptanalysis with C++20: Creating and Programming Advanced Algorithms

Marius Iulian Mihailescu
Bucharest, Romania

Stefania Loredana Nita
Bucharest, Romania

ISBN-13 (pbk): 978-1-4842-6585-7
ISBN-13 (electronic): 978-1-4842-6586-4
https://doi.org/10.1007/978-1-4842-6586-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Editorial Operations Manager: Mark Powers

Cover designed by eStudioCalamar

Cover image by Devin Avery on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484265857. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To our families.

Table of Contents

About the Authors

Marius Iulian Mihailescu, PhD is the CEO of Dapyx Solution Ltd., a company based in Bucharest, Romania. He is involved in information security- and cryptography-related research projects. He is a lead guest editor for applied cryptography journals and a reviewer for multiple publications on information security and cryptography profiles. He has authored and co-authored more than 30 articles for conference proceedings, 25 articles for journals, and four books. For more than six years he has served as a lecturer at well-known national and international universities (University of Bucharest, Titu Maiorescu University, Spiru Haret University of Bucharest, and Kadir Has University, Istanbul, Turkey). He has taught courses on programming languages (C#, Java, C++, Haskell) and object-oriented system analysis and design with UML, graphs, databases, cryptography, and information security. He worked for three years as an IT Officer at Royal Caribbean Cruises Ltd. where he dealt with IT infrastructure, data security, and satellite communications systems. He received his PhD in 2014 and his thesis is on applied cryptography over biometrics data. He holds MSc degrees in information security and software engineering.

Stefania Loredana Nita, PhD is a software developer and researcher at the Institute for Computers of the Romanian Academy. Her PhD thesis is on advanced cryptographic schemes using searchable encryption and homomorphic encryption. At the Institute for Computers, she works on research and development projects that involve searchable encryption, homomorphic encryption, cloud computing security, Internet of Things, and big data. She worked for more than two years as an assistant lecturer at the University of Bucharest where she taught courses on advanced programming techniques, simulation methods, and operating systems. She has authored and co-authored more than 25 workpapers for conferences and journals, and has co-authored four books. She is a lead guest editor for special issues on information security and cryptography such as *Advanced Cryptography and Its Future: Searchable and Homomorphic Encryption.* She has an MSc degree in software engineering and BSc degrees in computer science and mathematics.

About the Technical Reviewer

Doug Holland is a Software Engineer and Architect at Microsoft Corporation. He holds a Master's degree in software engineering from the University of Oxford. Before joining Microsoft, he was awarded the Microsoft MVP and Intel Black Belt Developer awards.

Acknowledgments

We would like to thank our editors for their support, our technical reviewer for his constructive comments and suggestions, the entire team that makes publishing this book possible, and last but not least, our families for their unconditional support and encouragement.

PART I

Foundations

CHAPTER 1

Getting Started in Cryptography and Cryptanalysis

Knowledge is one of the most important aspects to consider when designing and implementing complex systems, such as companies, organizations, military operations, and so on. Information falling into the wrong hands can be a tragedy and can result in a huge loss of business or disastrous outcomes. To guarantee the security of communications, cryptography can be used to encode information in such a way that nobody will be able to decode it without having the legal right. Many ciphers have been broken when a flaw has been found in their design or enough computing power has been applied to break an encoded message. Cryptology, as you will see later, consists of *cryptography* and *cryptanalysis*.

With the rapid evolution of electronic communication, the number of issues raised by information security significantly increases every day. Messages that are shared over publicly accessible computer networks around the world must be secured and preserved and must get the proper security mechanisms to protect against abuse. The business requirement in the field of electronic devices and their communication consists of having digital signatures that can be recognized by law. Modern cryptography provides solutions to all these problems.

The idea for this book started from experiences in several directions: (1) cryptography courses for students (at the graduate and undergraduate levels) in computer science at the University of Bucharest and Titu Maiorescu University; (2) industry experience achieved at national and international companies; (3) ethical hacking best practices; and (4) security audits. The goal of this book is to present the most advanced cryptography and cryptanalysis techniques together with their

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_1

implementations using C++20. This book will offer a practical perspective, giving the readers the necessary tools to design cryptography and cryptanalysis techniques in terms of practice. Most of the implementations are in C++20 using the latest features and improvements of the programming language (see Chapter 6). The book is an advanced and exhaustive work, offering a comprehensive view of the most important topics in information security, cryptography, and cryptanalysis. The content of the book can be used in a wide spectrum of areas by many professionals, such as security experts with their audits, military experts and personnel, ethical hackers, teachers in academia, researchers, software developers, software engineers when security and cryptographic solutions must be implemented in a real business software environment, professors teaching student courses (undergraduate and graduate level, master's degree, professional and academic doctoral degree), business analysts, and many more.

Cryptography and Cryptanalysis

It is very important to understand the meanings of the main concepts involved in a secure communication process and to know their boundaries.

- *Cryptology* is the science or art of secret writings. The main goal is to protect and defend the secrecy and confidentiality of the information with the help of cryptographic algorithms.
- *Cryptography* is the defensive side of cryptology. The main objective is to create and design cryptographic systems and their rules. Cryptography is a special kind of art, the art of protecting information by transforming it into an unreadable format called ciphertext.
- *Cryptanalysis* is the offensive side of cryptology. Its main objective is to study the cryptographic systems to provide the necessary characteristics to fulfill the function for which they have been designed. Cryptanalysis can analyze the cryptographic systems of third parties through cryptograms to break them in order to obtain useful information for business purposes. Cryptanalysts, code breakers, and ethical hackers are the people who deal with the field of cryptanalysis.

- *A cryptographic primitive* represents a well-established or low-level cryptographic algorithm that is used to build cryptographic protocols. Examples of such routines include hash functions or encryption functions.

The book provides a deep examination of all three aspects from a practical point of view with references to the theoretical background by illustrating how a theoretical algorithm should be analyzed for implementation.

Book Structure

The book is divided into 23 chapters in three parts (see Table 1-1): Part I: Foundations (Chapters 1-8), Part II: Pro Cryptography (Chapters 9-16), and Part III: Pro Cryptanalysis (Chapters 17-23). Figure 1-1 shows how to read the book and what chapters depend on each other.

The ***Part I: Foundations (Chapters 1-8)*** covers, from a beginner to advanced level and from theoretical to practical, the basic concepts of cryptography (*Chapter 2*). *Chapter 3* covers a collection of key elements regarding complexity theory, probability theory, information theory, number theory, abstract algebra, and finite fields and how they can be implemented using C++20, showing their interaction with the cryptography and cryptanalysis algorithms.

Chapters 4 and *Chapter 5* focus on integer arithmetic and floating-point arithmetic processing. These chapters are vital because other chapters and algorithms depend on the content of these chapters. Number representations and working with them via the memory of the computer can be a difficult task.

In *Chapter 6*, we discuss the newest features and enhancements of C++20. We give a presentation on how the new features and enhancements play an important role in developing cryptography and cryptanalysis algorithms and methods. We cover three-way comparisons, lambdas in unevaluated contexts, string literals, atomic smart pointers, <version> headers, ranges, coroutines, modules, and more.

Chapter 7 presents the most important guidelines for securing the coding process, keeping an important balance between security and usability based on the most expected scenarios based on trusted code. We cover important topics such as securing state data, security and user input, security-neutral code, and library codes that expose the protected resources.

Chapter 8 introduces the cryptography model and services that are used by C++. We cover important topics like C++ basic implementations, object inheritance, how cryptography algorithms are implemented, stream design, configuring cryptography classes, how to choose cryptography algorithms, generating the keys for encryption and decryption, storing asymmetric keys in a key container, cryptographic signatures, ensuring data integrity using hash codes and functions, creating and designing cryptographic schemes, encryption of XML elements with symmetric keys, assuring and guaranteeing interoperability of the applications between different platforms, such as Windows, MacOS, UNIX/Linux, and more.

Part II: Pro Cryptography (Chapters 9-16) contains the most important modern cryptographic primitives. *Chapters 9-16* discuss the advanced cryptography topics by showing implementations and how to approach this kind of advanced topic from a mathematical background to a real-life environment.

Chapter 9 discusses Cryptography Next Generation (CNG), which is used in the implementation of the Elliptic Curve Diffie-Hellman (ECDH) algorithm, and how to realize the necessary cryptographic operations.

Chapter 10 provides an introduction to the Lattice Cryptography Library and how it works, pointing out the importance of post-quantum cryptography. Implementations of key exchange protocols proposed by Alkim, Ducas, Poppelmann, and Schwabe [1] are discussed. We continue our discussion with an instantiation of Chris Peikert's key exchange protocol [2]. We point out that the implementation is based on modern techniques for computing, known as the number theoretic transform (NTT). The implementations apply errorless fast convolution functions over successions of integer numbers.

Chapter 11 and *Chapter 12* present two important cryptographic primitives, homomorphic and searchable encryption. For searchable encryption (SE), *Chapter 11* presents an implementation using C++20 and showing the advantages and disadvantages by removing the most common patterns from encrypted data. In *Chapter 12*, we discuss how to use the SEAL library for fully homomorphic encryption. The implementation is discussed based on the proposal of Shai Halevi and Victor Shoup in [3].

Chapter 13 covers the issues that are generated during the implementation of (ring) learning with errors cryptography mechanisms. We give as an example an implementation of the lattice-based key exchange protocol, a library that is used only for experiments.

Chapter 14 is based on the new concepts behind chaos-based cryptography and how they can be translated into practice. The chapter generates some new outputs and its contribution is important for the advancement of cryptography as it is a new topic that hasn't received proper attention until now.

Chapter 15 discusses new methods and their implementations for securing big data environments, big data analytics, access control methods (key management for access control), attributed-based access control, secure search, secure data processing, functional encryption, and multi-party computation.

Chapter 16 points out the security issues raised by applications that run in a cloud environment and how they can be resolved during the designing and implementation phase.

In ***Part III: Pro Cryptanalysis (Chapters 17-23)***, we deal with advanced cryptanalysis topics and we show how to pass the barrier between theory and practice, and how to think of cryptanalysis in terms of practice by eliminating the most vulnerable and critical points of a system or software application in a network or distributed environment.

Starting with *Chapter 17* we provide an introduction to cryptanalysis by presenting the most important characteristics of cryptanalysis.

Chapter 18 shows the important criteria and standards used in cryptanalysis, how the tests of cryptographic systems are made, the process of selecting cryptographic modules, cryptanalysis operations, and classifications of cryptanalysis attacks.

In *Chapter 19* and *Chapter 20*, we show how to implement and design linear and differential and integral cryptanalysis. We focus on techniques and strategies where the primary role is to show how to implement scripts for attacking linear and differential attacks.

Chapter 21 presents the most important attacks and how they can be designed and implemented using C++20. You study the behavior of software applications when they are exposed to different attacks and you exploit the source code. We also discuss software obfuscation and show why this is a critical aspect that needs to be taken into consideration by the personnel involved in implementing the process of the software. Also, we show how this analysis can lead to machine learning and artificial intelligence algorithms that can be used to predict future attacks against software applications that are running in a distributed or cloud environment.

In *Chapter 22*, we go through the text characterization methods and implementations. We discuss chi-squared statistics; identifying unknown ciphers; index of coincidence; monogram, bigram, and trigram frequency counts; quad ram statistics as a fitness measure; unicity distance; and word statistics as a fitness measure.

Chapter 23 presents the advantages and disadvantages of implementing the cryptanalysis methods, why they should have a special place when applications are developed in distributed environments, and how the data should be protected against such cryptanalysis methods.

Table 1-1. *Book Structure*

Part	Chapter #	Chapter Title
Part I Foundations (Foundational Topics)	1	Getting Started in Cryptography and Cryptanalysis
	2	Cryptography Fundamentals
	3	Mathematical Background and Its Applicability
	4	Large Integer Arithmetic
	5	Floating-Point Arithmetic
	6	New Features in C++20
	7	Secure Coding Guidelines
	8	Cryptography Libraries in C/C++20
Part II Pro Cryptography	9	Elliptic-Curve Cryptography
	10	Lattice-Based Cryptography
	11	Searchable Encryption
	12	Homomorphic Encryption
	13	Ring Learning with Errors Cryptography
	14	Chaos-Based Cryptography
	15	Big Data Cryptography
	16	Cloud Computing Cryptography
Part III Pro Cryptanalysis	17	Getting Started with Cryptanalysis
	18	Cryptanalysis Attacks and Techniques
	19	Linear and Differential Cryptanalysis
	20	Integral Cryptanalysis
	21	Brute Force and Buffer Overflow Attacks
	22	Text Characterization
	23	Implementation and Practical Approach of Cryptanalysis Methods

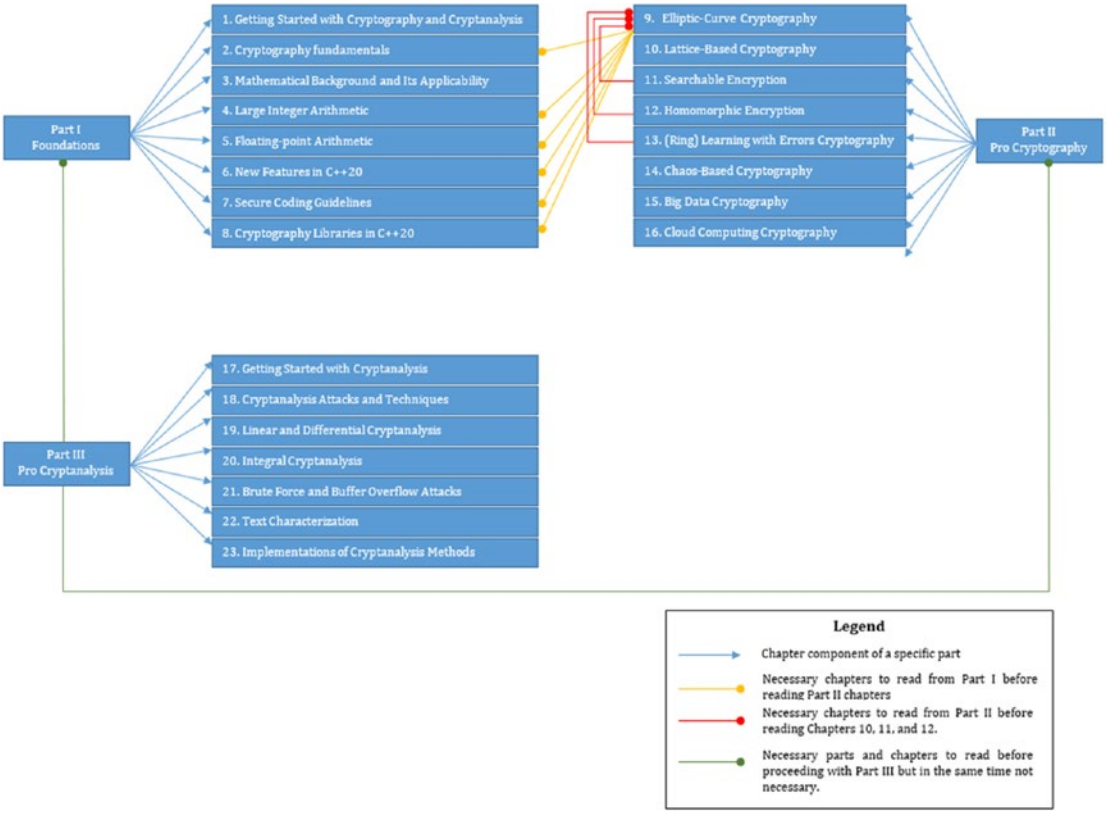


Figure 1-1. *A roadmap for readers and professionals*

Internet Resources

The Internet offers a significant amount of resources that are very useful regarding the topics in this book. These resources will help you keep up with progress in the fields:

- Bill's Security Site, `https://asecuritysite.com/`, contains various implementations of cryptographic algorithms. The website is created and updated by Bill Buchanan, a professor at the School of Computing at Edinburgh Napier University.
- Books by William Stallings [4] such as *Cryptography and Network Security*, (`http://williamstallings.com/Cryptography/`). His website contains an important set of tools and resources that he regularly updates, keeping in step with the most important advances in the field of cryptography.

- Schneier on Security, www.schneier.com/, contains sections on books, essays, accurate news, talks, and academic resources.

Forums and Newsgroups

Many USENET (quite deprecated but still very useful) newsgroups are dedicated to the important aspects of cryptography and network security. The most important are as follows:

- `sci.crypt.research` is one of the best groups for information about research ideas. It is a moderated newsgroup and its main purpose is to deal with research topics. Most of the topics are related to the technical aspects of cryptology.
- `sci.crypt` offers general discussions about cryptology and related topics.
- `sci.crypt.random-numbers` offers discussions about random number generators.
- `alt.security` offers general discussions on security topics.
- `comp.security.misc` offers general discussions on computer security topics.
- `comp.security.firewalls` offers discussions about firewalls and other related products.
- `comp.security.announce` covers CERT news and announcements.
- `comp.risks` offers discussions about the public risks from computers and users.
- `comp.virus` offers moderated discussions about computer viruses.

Also, there are many forums that deal with cryptography topics and news. The most important are as follows:

- **Reddit**: Cryptography News and Discussions [5]: The forum group contains general information and news about different topics related to cryptography and information security.

- **Security forums [6]**: They cover vast topics and discussions about computer security and cryptography.
- **TechGenix: Security [7]**: One of the most updated forums with news related to cryptography and information security. The group is maintained by world-leading security professionals.
- **Wilders Security Forums [8]**: The forum contains discussions and news about the vulnerabilities of software applications due to bad implementations of cryptographic solutions.
- **Security Focus [9]**: The forum contains a series of discussions about vulnerabilities raised by the implementations of cryptographic algorithms.
- **Security InfoWatch [10]**: The discussions are related to data and information loss.
- **TechRepublic: Security [11]**: The forum contains discussions about practical aspects and methodologies that can be used when software applications are designed and implemented.
- **Information Security Forum [12]**: A world-leading forum in the field of information security and cryptography. The forum contains conferences plus hands-on and practical tutorials for solving solutions for security and cryptographic issues.

Standards

Many of the cryptographic techniques and implementations described in this book follow the below standards. These standards have been developed and designed to cover the management practices and the entire architecture of security mechanisms, strategies, and services.

The most important standards covered by this book are as follows:

- **National Institute of Standards and Technology (NIST)**: NIST is the US federal agency that deals with standards, science, and technologies related to the US government. Excepting the national goal, the NIST Federal Information Processing Standards (FIPS) and the Special Publications (SP) have a very important worldwide impact.

- **Internet Society**: ISOC is one of the most important professional membership societies, with organizational and individual members worldwide. The society provides leadership on the issues that confront the future perspective of the Internet and applications that are developed using security and cryptographic mechanisms, with respect for the responsible groups, such as the Internet Engineering Task Force (IETF) and the Internet Architecture Board (IAB).
- **ITU-T**: The International Telecommunication Union (ITU) is one of the most powerful organizations within the United Nations system. It works with governments and the private sector to coordinate and administer the global telecom networks and services. ITU-T represents one of the three sectors of ITU. The mission of ITU-T consists of the production of the standards that cover all fields of telecommunications. The standards proposed by ITU-T are known as *recommendations.*
- **ISO**: The International Organizations for Standardization is a worldwide federation that contains national standards bodies from over 140 countries. The ISO is a nongovernmental organization that promotes the development of standardization and activities to facilitate the international exchange of services to develop cooperation with intellectual, scientific, and technological activities. The results of the ISO are international agreements published as international standards.

Conclusion

The era in which we are living is one of unimaginable evolution and incredible technologies that enable the instant flow of information at any time and any place. The secret consists of the convergence of the computer with networks; this forces the evolution and development of these incredible technologies from behind.

In this first chapter, we discussed the objectives of the book and its benefits. We explained the mission of the book, which is to address the practical aspects of cryptography and information security and its main intention in using the current work. The systems built upon advanced information technologies have a deep impact on our lives every day. These technologies are proving to be pervasive and ubiquitous.

The book represents the first practical step of translating the most important theoretical cryptography algorithms and mechanisms into practice through one of the most powerful programming languages, C++20.

In this chapter, you learned the following:

- The differences between cryptography, cryptanalysis, and cryptology.
- The structure of the book in order to help you follow the content easier. A roadmap was introduced in order to show the dependencies of each chapter. Each chapter was presented in detail, pointing out the main objective.
- A list of newsgroups, websites, and USENETs resources was covered in order provide sources for the latest news about cryptography and information security.
- The most significant standards for cryptography and information security were presented. You will get used to the workflow of each standard.

References

[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope" in *25th {USENIX} Security Symposium ({USENIX} Security 16)* (pp. 327-343). 2016.

[2] C. Peikert, "Lattice cryptography for the internet" in *International workshop on post-quantum cryptography* (pp. 197-219). Springer, Cham. October, 2014.

[3] S. Halevi and V. Shoup*, Design and implementation of a homomorphic-encryption library*. IBM Research (Manuscript), 6, 12-15. 2013.

[4] W. Stallings, *Cryptography and Network Security - Principles and Practice*, fifth edition. 2010: Pearson. 744.

[5] Reddit. Cryptography News and Discussions. Available from: www.reddit.com/r/crypto/.

[6] Forums, Security. Available online: www.security-forums.com/index.php?sid=acc302c71bb3ea3a7d631a357223e261.

[7] TechGenix, Security. Available online: http://techgenix.com/security/.

[8] Wilders Security Forums. Available online: www.wilderssecurity.com/.

[9] Security Focus. Available online: www.securityfocus.com/.

[10] Security InfoWatch. Available online: https://forums.securityinfowatch.com/.

[11] TechRepublic – Security. Available online: www.techrepublic.com/forums/security/.

[12] Information Security Forum. Available online: www.securityforum.org/.

CHAPTER 2

Cryptography Fundamentals

Cryptographic history is incredibly long and fascinating. A great and comprehensive reference is *The Code Book: The Secrets Behind Codebreaking* [1] published in 2003, which provides a non-technical history of cryptography. In the book, the story of cryptography begins around 2000 BC, when the Egyptians used it for the first (known) time, and ends with our era. It presents the main aspects of cryptography and hiding information for each period that is covered and describes cryptography's great contribution to both World Wars. Often, the art of cryptography is correlated with diplomacy, the military, and governments because its purpose is to keep safe sensitive data such as strategies or secrets regarding national security.

A crucial development in modern cryptography is the workpaper *New Directions in Cryptography* [2] proposed by Diffie and Hellman in 1976. The paper introduced a notion that changed how cryptography was seen: public-key cryptography. Another important contribution in this paper was the innovative way of exchanging keys; the security of the presented technique was based on the hardness assumption (basically, through the hardness assumption we refer to a problem that cannot be solved efficiently) of the discrete logarithm problem. Although the authors did not propose a practical implementation for their public-key encryption scheme, the idea was presented very clearly and started to get attention from the international cryptography community.

The first implementation of a public-key encryption scheme was made in 1978 by Rivest, Shamir, and Adleman, who proposed and implemented their encryption scheme, known nowadays as RSA [3]. The hardness assumption in RSA is the factoring of large integers. By looking in parallel at integer factorization for RSA and Shor's algorithm, note that Shor's Algorithm will run in polynomial time for quantum computers. This represents a significant challenge for any cryptographer who is using the hardness

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_2

assumption for factoring large integers. The increasing applications and interest in the factoring problem led to new factoring techniques. Important advances in this area were made in 1980, but none of the proposed techniques brought improvements to the security of RSA. Another important class of practical public-key encryption schemes was designed by ElGamal [4] in 1985. It was based on the hardness assumption of the discrete logarithm problem.

Another crucial contribution to public-key cryptography was the digital signature, which was adopted by the international standard ISO/IEC 9796 in 1991 [5]. The basis of the standard is the RSA public-key encryption scheme. A powerful scheme for digital signatures based on the discrete logarithm hardness assumption is the Digital Signature Standard, adopted by the United States Government in 1994.

Nowadays, the trends in cryptography include designing and developing new public-key schemes, adding improvements to the existing cryptographic mechanisms, and elaborating on security proofs.

The objective of this book is to provide a view of the latest updates of the principles, techniques, algorithms, and implementations of the most important aspects of cryptography in practice. We will focus on the practical and applied aspects of cryptography. You will be warned about difficult subjects and those that present issues. You will be guided to a proper references where you will find best practices and solutions. Most of the aspects presented in the book will be followed by implementations. This objective is to not obscure the real nature of cryptography. The book offers strong material for both implementers and researchers. The book describes algorithms and software systems with their interactions.

Information Security and Cryptography

In this book, we refer to the term and concept of *information* as to *quantity*. To go through the introduction to cryptography and to show its applicability in presenting algorithms and implementation technologies (such as C++), first you need to have a background in the issues that occur often in information security. When a particular transaction occurs, all parties involved in that transaction must be sure (or ensured) that specific objectives related to information security are met. A list of these security objectives is given in Table 2-1.

To define the issues regarding information security when the information is sent in a *physical format* (for example, documents), several protocols and security

mechanisms have been proposed. The objectives regarding information security may be accomplished by applying mathematical algorithms or work protocols on information that needs to be protected and additionally by following specific procedures and laws. An example of physical document protection is a sealed envelope (the mechanism of protection) that covers the letter (the information that needs to be protected) delivered by an authorized mail service (the trusted party). In this example, the protection mechanism has its limitations, but the technical framework has rigorous rules, through which any entity that opens the envelope without the right to so may be punished. There are situations in which the physical paper itself, which contains the information that needs to be protected, may have special characteristics that certify the originality of the data/information. For example, to refrain from the forging of bank notes, paper currency has special ink and matter.

Table 2-1. *Security Objectives*

Security Objective	Description
Privacy/confidentiality	The information is kept secret against unauthorized entities.
Signature	A technique that binds a signature by an entity (for example, a document)
Authorization	The action of authorizing an entity to do or to be something, in order to send the information between the sender and the receiver
Message authentication	The process/characteristic through which the origin of the data is authenticated; another meaning is corroboration of the information source.
Data integrity	The information is kept unaltered through techniques that keep away unauthorized entities or unknown means.
Entity authentication/ identification	The action of validating the identity of an entity, which may be a computer, person, credit card, etc.
Validation	The action of making available a (limited) quantity of time for authorization for using or manipulating the data or resources
Certification	The process of confirming the information by a trusted party, or acknowledgment of information by a trusted certification
Access control	The action of restricting access to resources to authorized parties
Timestamping	Metadata that stamps the time of creation or the existence of information

(*continued*)

Table 2-1. (*continued*)

Security Objective	Description
Witnessing	The action of validating the creation/existence of the information, made by an entity that is not the creator of the data
Receipt	The action of confirming the receiving of the information
Ownership	The action of giving to an entity the legal rights to use or transfer a particular information/resource
Confirmation	The action of validating the fact that certain services have been accomplished
Revocation	The action of withdrawing certification or authorization
Non-repudiation	The process of restraining the negation of other previous commitments or actions
Anonymity	The action of making anonymous an entity's identity that is involved in a particular action/process

From a conceptual point of view, how the information is manipulated did not change overmuch. We are considering here storing, registering, interpreting, and recording data. However, a manipulation that changed significantly is copying and modifying the information. An important concept in information security is the *signature*, which represents the foundation for more processes, such as non-repudiation, data origin authentication, identification, and witnessing.

To achieve the security of information in electronic communication, the requirements introduced by the legal and technical skills should be followed. On the other hand, it is not guaranteed that the above objectives of protection are fulfilled accordingly. The technical part of the information security is assured by cryptography.

Cryptography is the field that studies the mathematical techniques and tools that are connected to information security such as confidentiality, integrity (data), authentication (entity), and the origin of the authentication. Cryptography not only provides the security of the information but also a specific set of techniques.

Cryptography Goals

From the security objectives presented in Table 2-1, the following represent a basis from which can be derived the others:

- Privacy/confidentiality (Definitions 2.5 and 2.8)
- Data integrity (Definition 2.9)
- Authentication (Definition 2.7)
- Non-repudiation (Definition 2.6)

We will explain each of the four objectives in detail:

- *Confidentiality* is a service that is used to protect the content of the information from unauthorized entities and access. The confidentiality is assured through different techniques, from the use of mathematical algorithms to physical protection, which scramble the data into an incomprehensible form.
- *Data integrity* is a service that prevents unauthorized alteration of the information. Authorized entities should have the capability to discover and identify unauthorized manipulation of data.
- *Authentication* is a service that has an important role when data or application is authenticated. It implies identification. The authentication process is applied on both extremities that use the data (for example, the sender and the receiver). The rule is that each involved party should identify itself in the communication process. It is very important that both parties that are involved in the communication process should declare to each other their identity (the parties could be represented by a person or a system). At the same time, some characteristics of the data should accompany the data itself, such as its origin, content, the time of creation/sending, etc. From this point of view, cryptography branches the authentication into two categories: authentication of the entity and authentication of the data origin. The data origin authentication leads to data integrity.

- *Non-repudiation* is a service that prevents denials of previous actions made by an entity. When a conflict occurs because an entity denies its previous actions, it will be resolved by an existing trusted third party that will show the actions made over data.

One of the main goals of cryptography is to fulfill the four objectives described above on both sides, theory and practice.

Cryptographic Primitives

During the book, we will present several fundamental cryptographic tools, called *primitives*. Examples of primitives are encryption schemes (Definitions 2.5 and 2.8), hash functions (Definition 2.9), and schemes for digital signatures (Definition 2.6). Figure 2-1 shows a schematic description of these primitives and the relation between them. We will use many of the cryptographic primitives during the book, and we will provide practical implementations every time we use them. Before using them in real-life applications, the primitives should be subjected to an evaluation process in order to check if the below criteria are fulfilled:

- **Level of security**: It is slightly difficult to quantify the level of security. However, it can be quantified as the number of operations made in order to accomplish the desired objective. The level of security is usually defined based on the superior bound given by the volume of work necessary to defeat the objective.
- **Functionality**: To accomplish security objectives, in many situations the primitives are combined. You need to be sure that they work properly.
- **Operation methods**: When the primitives are used, they need different inputs and have different ways of working, resulting in different characteristics. In these situations, the primitives provide very different functionality that will depend on the mode of operation.
- **Performance**: This concept is related to the efficiency that a primitive can achieve in a specific and particular mode of operation.
- **Ease of implementation**: This concept is more a process than a criterion, and it refers to the primitive be used in practice.

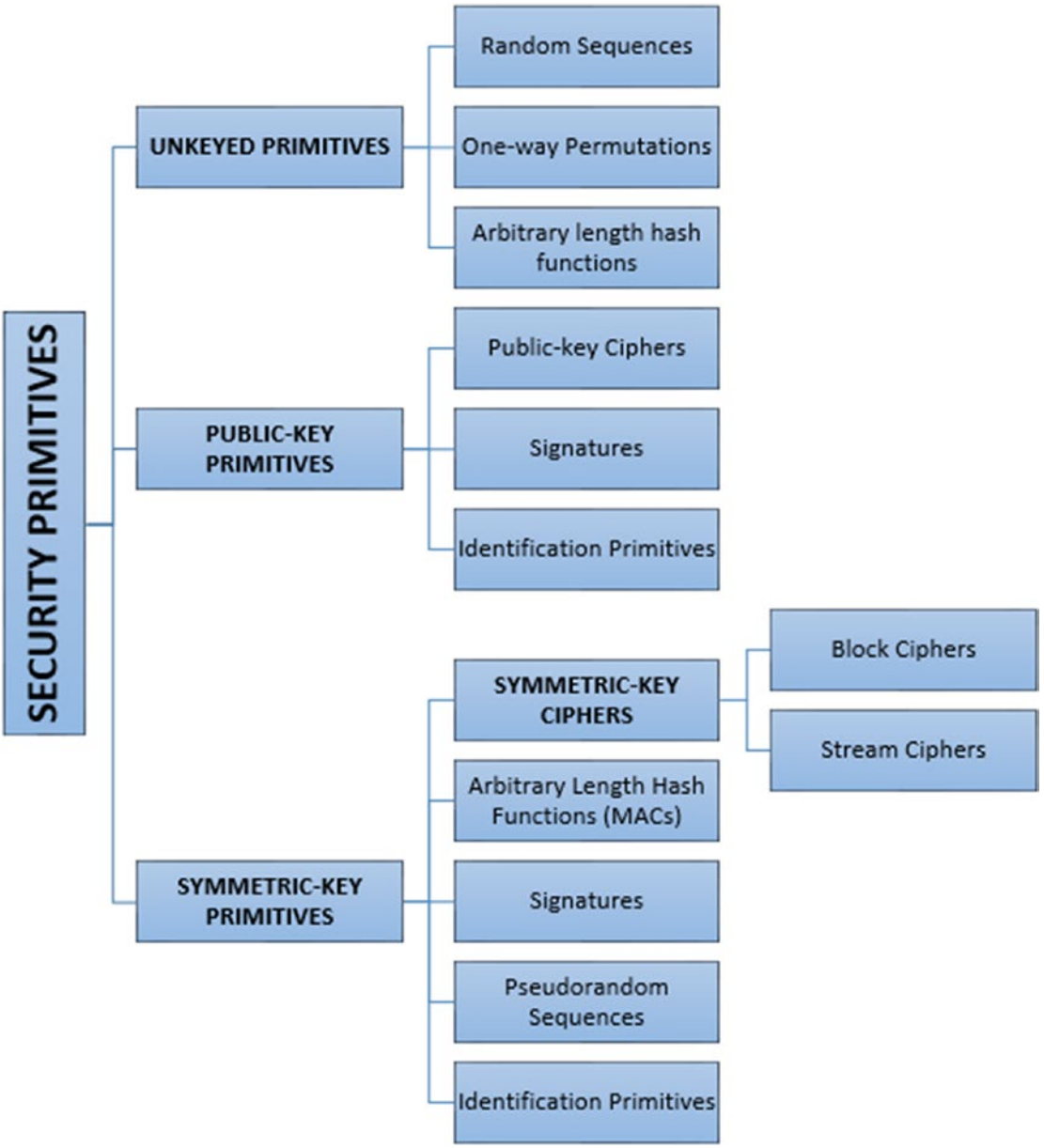


Figure 2-1. *Cryptographic primitives taxonomy*

The application and the available resources give importance to each of the above criteria.

Cryptography may be seen as an art practiced by professionals and specialists who proposed and developed ad-hoc techniques whose purpose was to fulfill important information security requirements. In the last few decades, cryptography has suffered

the transition from art to science and discipline. Nowadays, there are dedicated conferences and events in many fields of cryptography and information security. There are also international professional associations, such as the International Association for Cryptologic Research (IACR), whose aim is to promote the best results of research in the area.

This book is about cryptography and cryptanalysis, implementing algorithms and mechanisms using C++ with respect to standards.

Background of Mathematical Functions

This goal of the *book IS NOT to be a monograph on abstract mathematics*. However, getting familiar with some of the fundamental mathematical concepts is necessary and will prove to be very useful in practical implementations. One of the most important concepts that is fundamental to cryptography is the *function* in the mathematical sense. A *function* is also known in the literature as *transformation* or *mapping*.

Functions: One-to-One, One-Way, Trapdoor One-Way

Let's consider as a concept a *set,* which is a distinct set of objects, which are known as *elements* of that specific set. The following example represents set A, which has the elements a, b, c, this being denoted as $A = \{a, b, c\}$.

Definition 2.1 [18]. *Cryptography* is defined as the study of the mathematical techniques that are related to the aspects of the information security such as confidentiality, integrity (data), authentication (entity), and authentication of the data origin.

Definition 2.2 [18]. Let's consider that two sets A and B and rule f are defining a *function.* The rule f will assign to each element in A an element in B. The set A is known as the *domain* that characterizes the function and B represents the *codomain.* If a represents an element from A, written as $a \in A$, the *image* of a is represented by the element in B with the help of rule f; the image b of a is noted by $b = f(a)$. The standard notation for a function f from set A to set B is represented as $f: A \rightarrow B$. If $b \in B$, then we have a preimage of b, which is an element $a \in A$ for which $f(a) = b$. The entire set of elements in B that have at least one preimage is known as the *image* of f, noted as $Im(f)$.

Example 2.3. (*function*) Let's consider sets $A = \{a, b, c\}$ and $B = \{1, 2, 3, 4\}$, and the rule f from A to B as being defined as $f(a) = 2$, $f(b) = 4$, $f(c) = 1$. Figure 2-2 shows the

representation of the two sets A, B and the function f. The preimage of element 2 is a. The image of f is $\{1, 2, 4\}$.

Example 2.4. (*function*) Let's consider set $A = \{1, 2, 3, \ldots\ldots, 10\}$ and consider f to be the rule that for each $a \in A$, $f(a) = r_a$, where r_a represents the remainder when a^2 is being divided by 11.

$$f(1) = 1 \; f(6) = 3$$

$$f(2) = 3 \; f(7) = 5$$

$$f(3) = 9 \; f(8) = 9$$

$$f(4) = 5 \; f(9) = 4$$

$$f(5) = 3 \; f(10) = 1$$

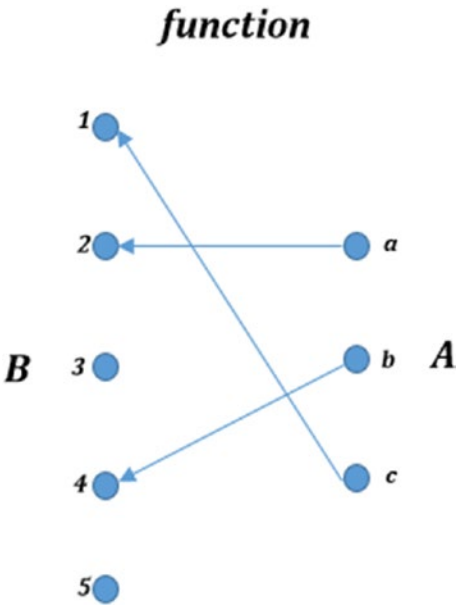


Figure 2-2. *Function f from a set A formed from three elements to a set B formed from five elements*

The image of f is represented by the set $Y = \{1, 3, 4, 5, 9\}$.

The scheme represents the main fundamental tool for thinking of a function (you can find it the literature as the *functional diagram*) as depicted in Figure 2-2, and each element from the domain A has precisely one arrow originated from it. For each element from codomain B we can have any number of arrows as being incident to it (including also zero lines).

Example 2.5. (*function*) Let's consider the following set defined as $A = \{1, 2, 3, \ldots, 10^{50}\}$ and consider the f to be the rule $f(a) = r_a$, where r_a represents the remainder in the case when a^2 is divided by $10^{50} + 1$ for all $a \in A$. In this situation, it is not feasible to write down f explicitly as in Example 2.4. This being said, the function is completely defined by the domain and the mathematical description that characterize the rule f.

One-to-One Functions

Definition 2.6 [18]. We will consider a function or transformation as $1 - 1$ (one-to-one) if each of the elements that can be found within the codomain B is represented as the image of at most one element in the domain A.

Definition 2.7 [18]. Let's consider that a function or transformation is *onto* if each of the elements found within the codomain B represents the image of at least one element that can be found in the domain. At the same time, function $f: A \to B$ is known as being onto if $Im(f) = B$.

Definition 2.8 [18]. Let's consider a function $f: A \to B$ to be considered $1 - 1$ and $Im(f) = B$. Then the function f is called *bijection.*

Conclusion 2.9 [18]. If $f: A \to B$ is considered $1 - 1$, then $f: A \to Im(f)$ represents the bijection. In special cases, if $f: A \to B$ is represented as $1 - 1$, and A and B are represented as finite sets with the same size, then f represents a bijection.

Using the scheme and its representation, if f is a bijection, then each element from B has exactly one line that is incidental with it. The functions in Examples 2.3 and 2.4 do not represent bijections. As you can see in Example 2.3, element 3 doesn't have the image of any other element that can be found within the domain. In Example 2.4, each element from the codomain is identified with two preimages.

Definition 2.10 [18]. If f is a bijection from A to B, then it is a quite simple matter to define a bijection g from B to A as follows: for each $b \in B$, we define $g(b) = a$ where $a \in A$ and $f(a) = b$. The function g is obtained from f and it is called *inverse function* of f and it denoted as $g = f^{-1}$.

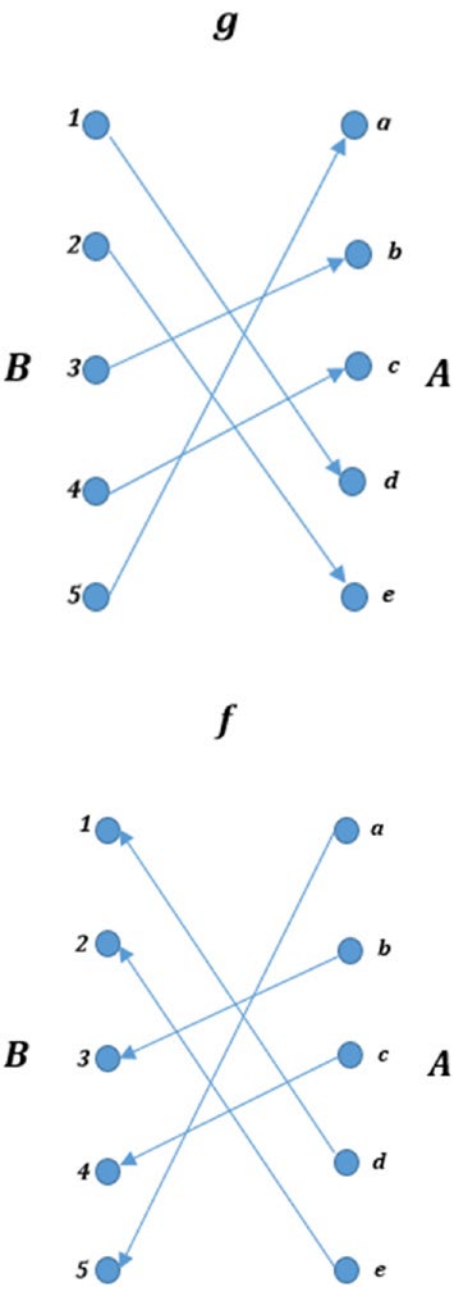


Figure 2-3. *Representation of bijection f and its inverse,* $g = f^{-1}$

Example 2.11. (*inverse function*) Let's consider sets $A = \{a, b, c, d, e\}$ and $Y = \{1, 2, 3, 4, 5\}$, and consider the rule f which is given and represented by the lines from

Figure 2-3. f represents a bijection and its inverse g is formed by reversing the sense of the arrows. The domain of g is represented by B and the codomain is A.

Note that if f is a bijection, f^{-1} is also a bijection. The bijections in cryptography are tools used for message encryption. The inverse transformations are used for decryption. The main condition for decryption is for transformations to be bijections.

One-Way Functions

In cryptography there are a certain types of functions that play an important role. Due to the rigor, a definition for a one-way function is given as follows.

Definition 2.12 [18]. Let's consider a function f from a set A to a set B that is called a *one-way* function if $f(a)$ proves to be simple and *easy* to be computed for all $a \in A$ but for "essentially all" elements $b \in Im(f)$ it is *computationally infeasible* to manage to find any $a \in A$ in such way that $f(a) = b$.

Note 2.13 [18]. This note represents some additional notes and clarifications of the terms used in Definition 2.12.

1. For the terms *easy* and *computationally infeasible,* a rigorous definition is necessary but it will distract attention from the general idea that is being agreed upon. For the goal of this chapter, the simple and intuitive meaning is sufficient.

2. The words *essentially all* stand for the idea that there are a couple of values $b \in B$ for which it is easy to find an $a \in A$ in such way that $b = f(a)$. As an example, one may compute $b = f(a)$ for a small number of a values and then for these, the inverse is known by a table look-up. A different way to describe this property of a one-way function is as follows: for any random $b \in Im(f)$ it is computationally feasible to have and find any $a \in A$ in such way that $f(a) = b$.

The following examples will show the concept behind a one-way function.

Example 2.14. (*one-way function*) Consider $A = \{1, 2, 3, \ldots, 16\}$ and let's define $f(a) = r_a$ for all the elements $a \in A$ where r_a represents the remainder when 3^x will be divided with 17.

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f(a)$	3	9	10	13	5	15	11	16	14	8	7	4	12	2	6	1

Let's assume that we have a number situated between 1 and 16. We can see that is very easy to find the image of it under f. Without having the table in front of you, for example, for 7 it is hard to find a given that $f(a) = 7$. If the number that you are given is 3, then is quite easy to see that $a = 1$ is what you actually need.

Remember that this example is focused on very small numbers. The key thing here is that the amount of effort to measure $f(a)$ is different than the amount of work in finding a given $f(a)$. Also for large numbers, $f(a)$ can be efficiently computed using the square-and-multiply algorithm [20], where the process of finding a from $f(a)$ is harder to find.

Example 2.15 [18]. (*one-way function*) A *prime number* is defined as a positive integer. The integer is bigger than 1 and its positive integers divisors are 1 and itself. Let's take into consideration primes p = 50633 and $q = 58411$, compute $n = pq = 50633 \cdot 58411 = 2957524163$, and consider $A = \{1, 2, 3, \ldots, n - 1\}$. We will define a function f on A by $f(a) = r_a$ for each $a \in A$, where r_a represents the remainder when x^3 is divided by n. For example, let's consider $f(2489991 = 1981394214$ since $2489991^3 = 5881949859 \cdot n + 1981394214$. Computing $f(a)$ represents a simple thing to be done, but reversing the procedure is quite difficult.

Trapdoor One-Way Functions

Definitions 2.16 [18]. A *trapdoor one-way function* is represented as a one-way function $f: A \rightarrow B$ with an extra property that by having information (also known as *trapdoor information)* it will be much more feasible to have an identification for any given $b \in Im(f)$, with an $a \in A$ in such way that $f(a) = b$.

Example 2.15 shows the concept of a trapdoor one-way function. With extra information about the factors of $n = 2957524163$ it becomes much easier to invert the function. The factors of 2957524163 are large enough that it would be difficult to identify them by hand calculation. We should be able to identify the factors very easily with the help of a computer program. For example, if we have very big, distinct prime numbers (each number has about 200 decimal digits) p and q, with the technology of today, finding p and q from n is very difficult even with the most powerful quantum computers. This is the well-known factorization problem known as *integer factorization problem*.

One-way and one-way trapdoor functions form the fundamental basis for public-key cryptography. These principles are very important, and they will become much clearer later when you explore the implementation of cryptographic techniques. It is vital and important to understand these concepts from this section because they are the main methods and the primary foundation for the cryptography algorithms implemented later in this chapter.

Permutations

Permutation represents functions that are in cryptographic constructs.

Definition 2.17 [18]. Consider S to be a finite set formed of elements. A *permutation* p on S represents a bijection as defined in Definition 2.8. The bijection is represented from S to itself, $p : S \to S$.

Example 2.18 [18]. This example represents a permutation example. Let's consider the following permutation: $S = \{1, 2, 3, 4, 5\}$. The permutation $p : S \to S$ is defined as

$$p(1) = 2, p(2) = 5, p(3) = 4, p(4) = 2, p(5) = 1$$

A permutation can be described in different ways. It can be written as above or as an array as in

$$p = \begin{pmatrix} 1 & 2 & 3\,4\,5 \\ 3 & 5 & 4\,2\,1 \end{pmatrix},$$

in which the top row in the array is represented by the domain and the bottom row is represented by the image under p as mapping.

Since the permutations are bijections, they have inverses. If the permutation is written as an away (second form), its inverse can be easily found by interchanging the rows in the array and reordering the elements from the new top row and the bottom row accordingly. In this case, the inverse of p is defined as follows:

$$p^{-1} = \begin{pmatrix} 1 & 2 & 3\,4\,5 \\ 5 & 4 & 1\,3\,2 \end{pmatrix}$$

Example 2.19 [18]. This example represents a permutation example. Let's consider A to be the set of integers $\{0, 1, 2, \ldots, p \cdot q - 1\}$ where p and q represent two distinct large primes. We need to suppose also that neither $p - 1$ nor $q - 1$ can be divisible by 3. The function $p(a) = r_a$, in which r_a represents the remainder when a^3 is divided by pq, can be demonstrated and shown as being the inverse perumutation. The inverse permutation is computationally infeasible by the computers of today, unless p and q are known.

Involutions

Involutions are known as functions having their own inverses.

Definition 2.20 [18]. Let's consider a finite set S and f defined as a bijection S to S, noted as $f: S \rightarrow S$. In this case, the function f will be noted as an *involution* if $f = f^{-1}$. Another way of defining this is $f(f(a)) = a$ for any $a \in S$.

Example 2.21 [18]. This example represents an involution case. Figure 2-4 shows an example of involution. Note that if j represents the image of i, then i represents the image of j.

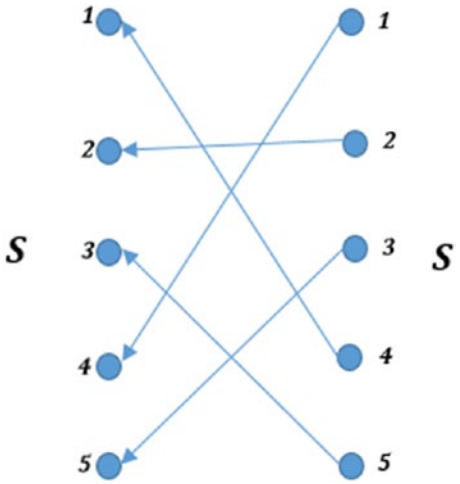


Figure 2-4. *Representation of an involution with an set S with five elements*

Concepts and Basic Terminology

It is very difficult to understand how cryptography was built using hard and abstract definitions when dealing with the scientific side of the field. In the following sections, we will list the most important terms and key concepts used in this chapter.

Domains and Codomains Used for Encryption

- $\mathcal{A}$ is shown as a finite set known as the *alphabet of definition*. We will consider as an example $\mathcal{A} = \{0,1\}$, which represents the binary alphabet, an alphabet frequently used as definition.
- $\mathcal{M}$ is a set known as the *message space*. The message space has the strings of symbols from an alphabet, $\mathcal{A}$. As an example, $\mathcal{M}$ may have binary strings, English text, French text, etc.

- $\mathcal{C}$ is the *ciphertext space*. $\mathcal{C}$ has strings of symbols from an alphabet, $\mathcal{A}$, which is totally different from the alphabet defined for $\mathcal{M}$. An element from $\mathcal{C}$ is called a ciphertext.

Encryption and Decryption Transformations

- The set $\mathcal{K}$ is called *key space*. The elements of $\mathcal{K}$ are called *keys*.
- For each $e \in \mathcal{K}$, there is a unique transformation E_e, representing a bijection from $\mathcal{M}$ to $\mathcal{C}$ (i.e., $E_e : \mathcal{M} \to \mathcal{C}$). E_e is called the *encryption function* or *encryption transformation*. If the encryption process is reversed, then E_e should be a bijection, such that each unique plain message is recovered from one unique ciphertext.

  For each $d \in \mathcal{K}$, there is a transformation D_d, representing a bijection from $\mathcal{C}$ to $\mathcal{M}$ (i.e., $D_d : \mathcal{C} \to \mathcal{M}$). D_d is called a *decryption function* or *decryption transformation*.
- The process of *encrypting the message* $m \in \mathcal{M}$ or the *encryption of m* consists of applying the transformation E_e over *m*.
- The process of *decrypting the ciphertext* $c \in \mathcal{C}$ or the *decryption of c* consists of applying the transformation D_d over *c*.
- An encryption scheme has two important sets: $\{E_e : e \in \mathcal{K}\}$, which represents the set of the encryption transformations, and $\{D_d : d \in \mathcal{K}\}$, which represents the set of the decryption transformations. The relationship between the elements of the two sets is the following: for each $e \in \mathcal{K}$ exists a unique key $d \in \mathcal{K}$ in such that $D_d = E_e^{-1}$; in other words, we have the relationship $D_d(E_e(m)) = m$ for all $m \in \mathcal{M}$. Another term for encryption schemes is *cipher*.
- In the above definition, the encryption key *e* and the decryption key *d* form a pair, usually noted as (*e*, *d*). In symmetric encryption schemes, *e* and *d* are the same, while in asymmetric (or public-key) encryption schemes they are different.

- To *construct* an encryption scheme, the following components are needed: the messages (or plain-texts) space $\mathcal{M}$, the cipher-space $\mathcal{C}$, the keys space $\mathcal{K}$, the set of encryption transformations $\{E_e : e \in \mathcal{K}\}$, and the set of decryption transformations $\{D_d : d \in \mathcal{K}\}$.

The Participants in the Communication Process

The components involved in the communication process are the following (Figure 2-5):

- The *entity (party)* is that component that works with the information: sending, receiving, manipulating it. The entities/parties from Figure 2-5 are *Alice*, *Bob*, and *Oscar*. However, in real applications, the entities are not necessarily people; they may be authorities or computers, for example.
- The *sender* is one of the entities of a two-party communication and it initiates the transmission of the data. The sender from Figure 2-5 is *Alice*.
- The *receiver* is the other entity of a two-party communication and it is the intended recipient of the information. The receiver from Figure 2-5 is *Bob*.
- The *communication channel* is the component through which the sender and the receiver communicate.
- The adversary is an unauthorized entity in a two-party communication and it is different from the sender and the receiver. Its objective is to break the security on the communication channel in order to access the information. Other terms for the adversary are[1]: enemy, attacker, opponent, eavesdropper, intruder, and interloper. It has different types (passive and active) and will behave differently according to aspects regarding the encryption scheme or its intentions. Often, the attacker clones and acts like the legitimate sender or the legitimate receiver.

[1]Alice and Bob. Available online: https://en.wikipedia.org/wiki/Alice_and_Bob

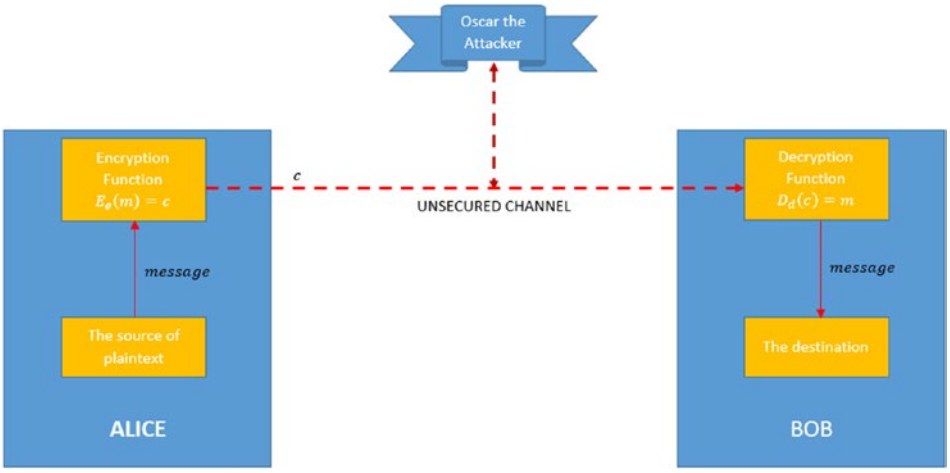


Figure 2-5. *Example of two-party communication process applying encryption*

Digital Signatures

In this book, another technique that we will work with is the digital signature. Digital signatures are very important in some processes, like authentication, authorization, or non-repudiation. The digital signature is used to map an individual's identity with a piece of information. When "something" is digitally signed, it means that the message and the confidential information owned by an individual are converted into a tag called a signature.

The components of the signing process are

- $\mathcal{M}$ is the set of messages that can be signed.
- $\mathcal{S}$ is the set of *signatures*. They can have a form of binary strings with a predefined length.
- $\mathcal{S}_A$ represents the transformation between $\mathcal{M}$ and $\mathcal{S}$, called a *signing transformation*, and it is made by entity *A*. The entity will keep $\mathcal{S}_A$ secret and will use it to sign messages from $\mathcal{M}$.
- V_A is the transformation between $\mathcal{M} \times \mathcal{S}$ to the set $\{true, false\}$. The Cartesian product $\mathcal{M} \times \mathcal{S}$ contains the pair of elements (m, s) where $m \in \mathcal{M}$ and $s \in \mathcal{S}$. The transformation V_A is public and it is used by different entities to check if the signatures were created by entity *A*.

Signing Process

Entity A, called the signer, creates a signature $s \in \mathcal{S}$ for a particular message $m \in \mathcal{M}$ following these steps:

- Compute $s = S_A(m)$.
- Transmit the pair (m, s) to the desired receiver.

Verification Process

When the receiver entity B wants to check if entity A created the signature s for the message m, it proceeds as follows:

- Obtain the verification function V_A for the entity A.
- Compute $u = V_A(m, s)$.
- If $u = true$, then the signature was created by entity A; if $u = false$, then the signature was not created by entity A.

Public-Key Cryptography

Public-key cryptography (PKC) has an important role in C++ when similar algorithms need to be incorporated. Many significant commercial libraries implement developer-specific public-key cryptography solutions, such as [21-30].

Next, you will see how the public-key cryptography works. For this, recall that $\mathcal{K}$ is the key space. Let's consider the set of the encryption transformations as $\{E_e : e \in \mathcal{K}\}$ and the set of the decryption transformations as $\{D_d : d \in \mathcal{K}\}$. Further, let's consider the pair of encryption and decryption transformations as (E_e, D_d), where E_e can be learned by anyone, for every e. From having E_e, determining D_d must be computationally unrealizable; in other words, from a random ciphertext $c \in \mathcal{C}$ it's impossible to find out the message $m \in \mathcal{M}$, such that $E_e(m) = c$. This property is strong and it means that the corresponding decryption key d (which must be secret/private) may not be computed/determined from either given e (which is public).

With the settings from above, take a look at Figure 2-6 and let's consider the communication channel between two parties, namely Alice and Bob.

- Bob chooses a pair of keys, (e, d).
- Bob makes the encryption key e publicly available, such that Alice can access it over any channel, and keeps secret and in safe the decryption key d. In the specialty literature, in PKC the encryption key is called *the public key* and the decryption key is called *the secret/private key*.
- When Alice wants to send a message $m \in \mathcal{M}$ to Bob, she uses Bob's public key e to determine the encryption transformation E_e, and then she applies it over m. Finally, Alice obtains the encryption $c = E_e(m) \in \mathcal{C}$ and sends it to Bob.
- When Bob wants to decrypt the encrypted message $c \in \mathcal{C}$ received from Alice, he uses his private key d to determine the transformation decryption D_d, and then he applies it over c. Finally, he obtains $m = D_d(c) \in \mathcal{M}$.

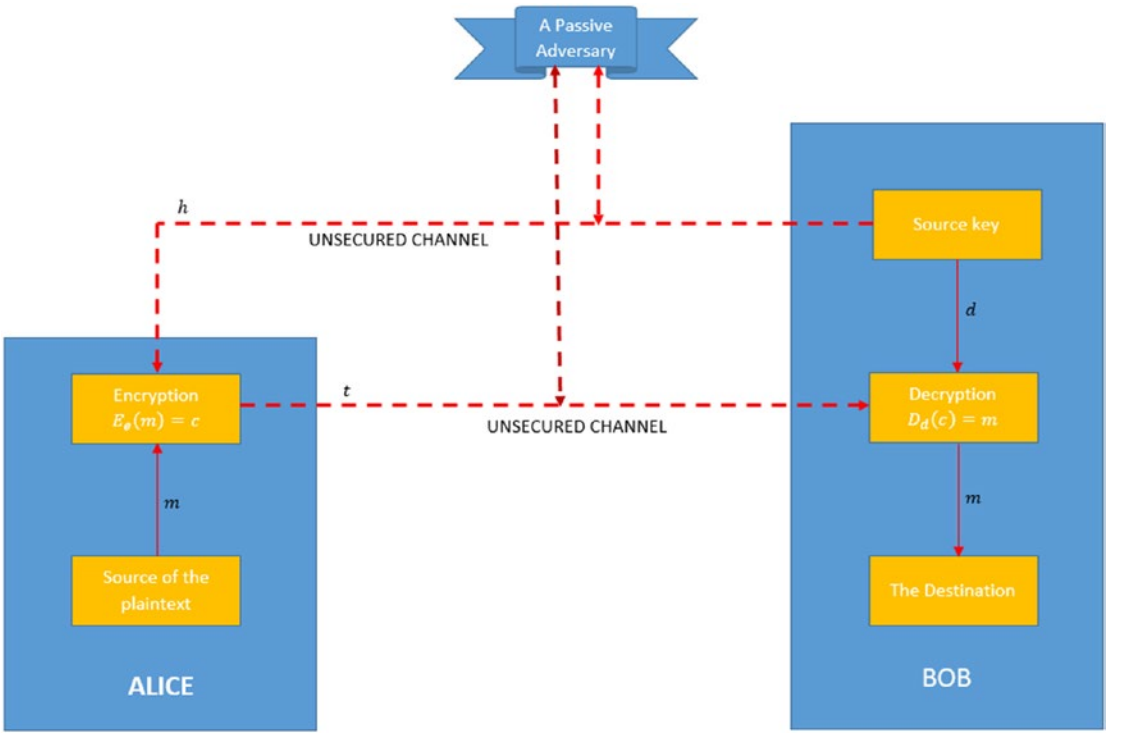


Figure 2-6. *The process of encryption using public-key mechanism*

There's no need to keep the encryption key e secret; it can be made public. Every individual can then send encrypted messages to Bob, which can be decrypted only by Bob. Figure 2-7 illustrates the idea, where A_1, A_2, and A_3 represents different entities. Remember if A_1 destroys the message m_1 after encrypting it to c_1, then even A_1 is in the position of not being able to recover m_1 from c_1.

Let's take the following analog example to make it simple. Consider a metal box with the cover secured by a lock with a particular combination. Bob is the only one who knows how to open the lock. If the lock stays open and is made accessible to the public for different purposes, someone can put a message inside and lock the lock.

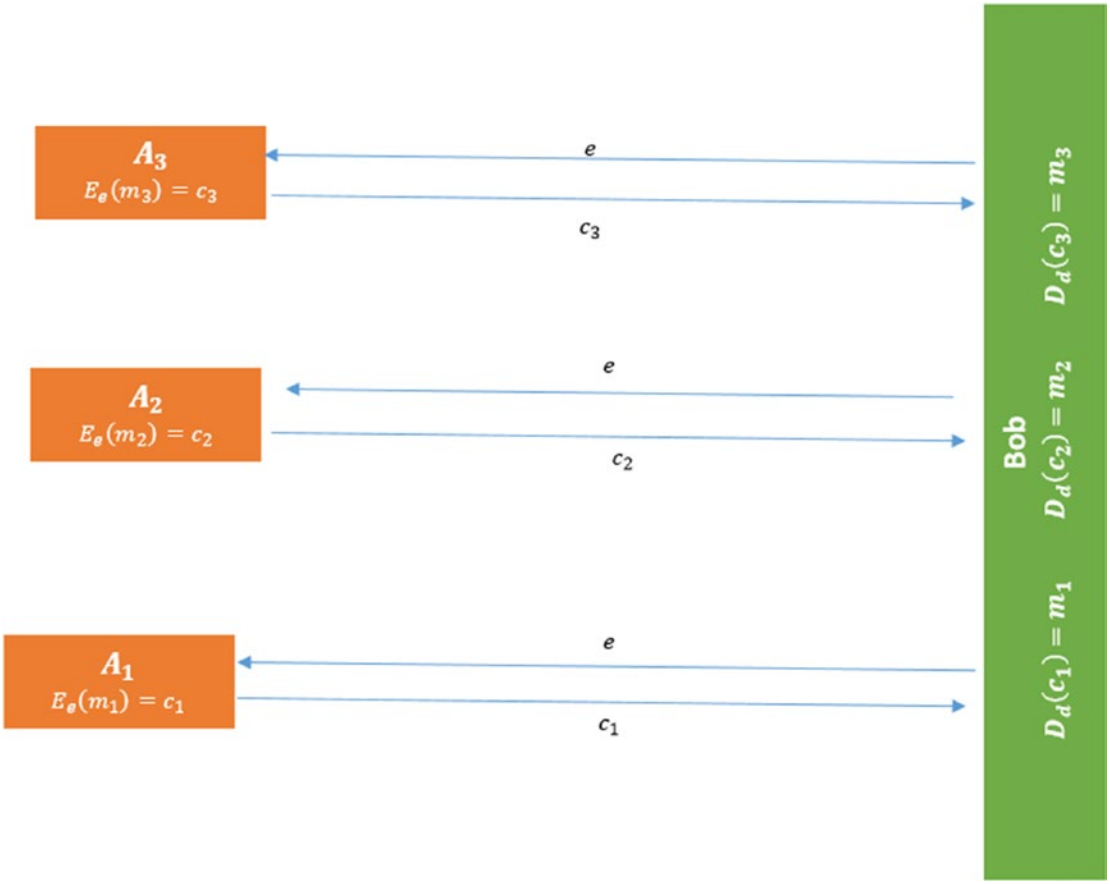


Figure 2-7. *How public-key encryption is used*

Hash Functions

Hash functions are one of the primary primitives in modern cryptography. Also known as a one-way hash function, a hash function represents a computationally efficient function that maps the binary string to binary strings with an arbitrary length with a fixed length known as hash values.

As an example of implementation of a hash function (SHA-256, see Figure 2-8), we will examine the following implementation in C++ using C++20 new features (see Listing 2-1). The implementation is done in accordance with the NIST Standard[2].

[2]NIST Standard for Hash Functions implementations, https://csrc.nist.gov/projects/hash-functions

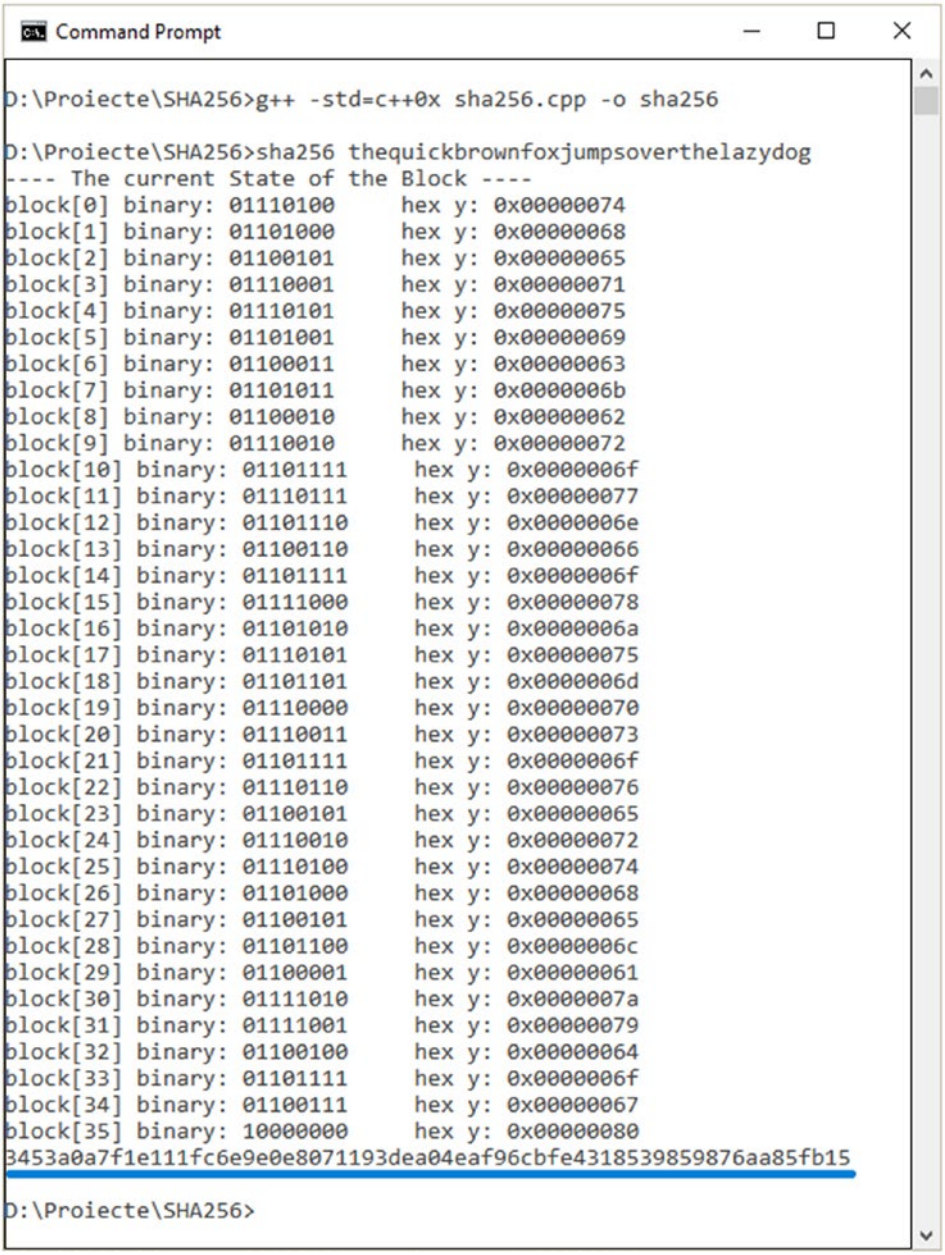


Figure 2-8. *Exemple of SHA-25 execution*

Listing 2-1. Source Code for Implementation of SHA256

```
#include <iostream>       //** standard input/output library
#include <sstream>        //** templates and types for interoperation
                          //** between flow buffers and string objects
#include <bitset>         //** storing bits library
#include <vector>         //** for representing arrays as containers
#include <iomanip>        //** for manipulation of the parameters
#include <cstring>        //** for manipulation of the strings

using namespace std;      //** for avoiding writing "std::"

//** ASCII string will be converted as a binary representation
vector<unsigned long> binaryConversion(const string);

//** for addings padding's to messages and making sure that they are
//** multiple of 512 bits
vector<unsigned long> addPadOf512Bits(const vector<unsigned long>);

//** We will change the n 8 bit blocks to 32 bits words
vector<unsigned long> resizingTheBlock(vector<unsigned long>);

//** will contain the actual hash value
string computingTheHash(const vector<unsigned long>);

//** variables and constants using during debugging
string displayAsHex(unsigned long);
void outputTheBlockState(vector<unsigned long>);
string displayAsBinary(unsigned long);
const bool displayBlockStateAddOne = 0;
const bool displayDistanceFrom512Bit = 0;
const bool displayResultsOfPadding = false;
const bool displayWorkVariablesForT = 0;
const bool displayT1Computation = false;
const bool displayT2Computation = false;
const bool displayTheHashSegments = false;
const bool displayWt = false;
```

```
//** defined in accordance with the NIST standard
#define ROTRIGHT(word,bits) (((word) >> (bits)) | ((word) << (32-(bits))))
#define SSIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SSIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))
#define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

//** in accordance with the latest updates of the NIST standard
//** we will replace BSIG0 with EP0 and BSIG1 with EP0 in our
//** implementation
#define BSIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define BSIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))

#define EP0(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
#define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))

//** we will verify if the process of checking (testing) is enabled
//** by the missed arguments in the command line.
//** The steps are as follows:
//** (1) Take the ascii string and convert it in n 8 bit segments by
//** representing the ascii value of each independently character
//** (2) add paddings to the message in order to get a 512 bit long
//** (3) take separately each 8 bit ascii value and convert it to 32
//** bit words and create a combination of them.
//** (4) calculate the hash and get the vallue
//** (5) if we are doing test, take the result and compare it with
//** expected result
int main(int argc, char* argv[])
{
      string theMessage = "";
      bool testing = 0;

      switch (argc) {
          case 1:
              cout << "There is no input string found. The test will
              be run using random first three letters abc.\n";
                  theMessage = "abc";
                  testing = true;
```

```
            break;
            case 2:
            if (strlen(argv[1]) > 55)
             {
                 cout << "The string provided is biger than 55
                 characters length. Enter a shorter string."
                         << " or message!\n";
                 return 0;
            }
              theMessage = argv[1];
              break;
              default:
              cout << "There are too many items in the command line.";
              exit(-1);
              break;
     }

//** storing all the blocks
     vector<unsigned long> theBlocksArray;

     //** convert the message to a vector of strings by hacving it
//** represented it as a 8 bit variable
     theBlocksArray = binaryConversion(theMessage);

     //** add padd to it in order to get a full of 512 bits long
     theBlocksArray = addPadOf512Bits(theBlocksArray);

      //** create a separate combination of the 8 bit segments into
      //** single 32 bits sections
      theBlocksArray = resizingTheBlock(theBlocksArray);

      //** compute the hash using computingTheHash function
      string myHash = computingTheHash(theBlocksArray);
```

```
     //** if testing is found on true the software app will execute
//** a self check by checking if the hash value computed for
//** "abc" is equal with the expected hash
     if (testing) {
          const string theCorrectHashForABC =
          "ba7816bf8f01cfea414140de5dae2223b00361a3961
          77a9cb410ff61f20015ad";
          if (theCorrectHashForABC.compare(myHash) != 0) {
               cout << "\tThe test didn't occur with success!\n";
               return(1); }
          else {
               cout << "\tTest has been done with success!\n";
               return(0); } }

     cout << myHash << endl;
     return 0;  }

//** the function purpose is to resize the blocks from 64 and 8 bit
//** to 16 and 32 bit sections. The function as input will take a
//** vector of individual 8 bit ascii values. As output we will get a
//** vector with 32 bit words that are found within a combination of
//** ascii values.
vector<unsigned long> resizingTheBlock(vector<unsigned long>
inputOf8BitAsciiValues)
{
     vector<unsigned long>
outputOf32BitWordsCombinedAsAsciiValues(16);

     //** parse all 64 sections using a 4 step and mergem them
//** accordingly
     for(int i = 0; i < 64; i = i + 4) {
          //** create for beginning a big 32 bit section first
          bitset<32> temporary32BitSection(0);
```

```
          //** create a shifting of the blocks on their assigned
//** positions
          temporary32BitSection = (unsigned long)
inputOf8BitAsciiValues[i] << 24;
temporary32BitSection |= (unsigned long)
inputOf8BitAsciiValues[i + 1] << 16;
          temporary32BitSection |= (unsigned long)
inputOf8BitAsciiValues[i + 2] << 8;
          temporary32BitSection |= (unsigned long)
inputOf8BitAsciiValues[i + 3];
          //** set the new 32 bit word within the proper output of
//** the array location
          outputOf32BitWordsCombinedAsAsciiValues[i/4] =
          temporary32BitSection.to_ulong(); }

     return outputOf32BitWordsCombinedAsAsciiValues; }

//** the function display the contents of all the blocks as binary
//** format. The function is used only for debugging purpose.
void outputTheBlockState(vector<unsigned long>
vectorOfCurrentBlocks) {
     cout << "---- The current State of the Block ----\n";
     for (int i = 0; i < vectorOfCurrentBlocks.size(); i++) {
          cout << "block[" << i << "] binary: " <<
displayAsBinary(vectorOfCurrentBlocks[i])
                << "     hex y: 0x" <<
displayAsHex(vectorOfCurrentBlocks[i]) << endl; }}

//** the function will display in hex format the content of the
//** blocks.
string displayAsHex(unsigned long input32BitBlock) {
     bitset<32> theBitSet(input32BitBlock);
     unsigned number = theBitSet.to_ulong();

     stringstream theStringStream;
     theStringStream << std::hex << std::setw(8) <<
std::setfill('0') << number;
```

```
     string temporary;
     theStringStream >> temporary;

     return temporary; }

//** the function will show the content of the blocks in hex. We are
//** using this function in order to avoid changing the stream from
//** hexa to dec and reversed as well.
string displayAsBinary(unsigned long input32OrLessBitBlock) {
     bitset<8> theBitSet(input32OrLessBitBlock);
     return theBitSet.to_string(); }

//** based on the string, it will take the entire set of the
//** characters and converts them into ascii binary.
vector<unsigned long> binaryConversion(const string
inputOfAnyLength) {
     //** the vector used to store all the ascii characters
          vector<unsigned long> vectorBlockHoldingAsciiCharacters;

     //** take each character and convert the ascii character to
//** the binary representation
     for (int i = 0; i < inputOfAnyLength.size(); ++i) {
          //** create a temporary variable. Use it to store the 8
//** bit template for ascii value
bitset<8> bitSetOf8Bits(inputOfAnyLength.c_str()[i]);

          //** the template of 8 bit add it into the block
vectorBlockHoldingAsciiCharacters.
push_back(bitSetOf8Bits.to_ulong());}

     return vectorBlockHoldingAsciiCharacters; }

//** get the ascii values stored as a vector in binary and add padding to
it in order to obtain a total of 512 bits.
vector<unsigned long> addPadOf512Bits(vector<unsigned long>
vectorBlockHoldingAsciiCharacters) {
     //** you can keep the variables names as given in the NIST
     //** for our implementation I have used my personal names for
//** variables in order to get a uniqueness of the code
```

```
//** the variable will store the length of the message in bits
int lengthOfMessageInBits = vectorBlockHoldingAsciiCharacters.size() * 8;

     int zeroesToAdd = 447 - lengthOfMessageInBits;

//** add another 8 bit block with the first bit being set to 1
     if(displayBlockStateAddOne)
          outputTheBlockState(vectorBlockHoldingAsciiCharacters);

     unsigned long t1Block = 0x80;
     vectorBlockHoldingAsciiCharacters.push_back(t1Block);

     if(displayBlockStateAddOne)
          outputTheBlockState(vectorBlockHoldingAsciiCharacters);
          outputTheBlockState(vectorBlockHoldingAsciiCharacters);

     //** we have 7 zeroes. We will need to substract 7 from
//** zeroesToAdd
     zeroesToAdd = zeroesToAdd - 7;

     //** debug mode. Find how much we need to get close to 512 bit
     if (displayDistanceFrom512Bit) {
          cout << "lengthOfMessageInBits = " <<
lengthOfMessageInBits << endl;
          cout << "zeroesToAdd = " << zeroesToAdd + 7 << endl; //
Plus 7 so this follows the paper. }

     //** debug mode
     if (displayDistanceFrom512Bit)
          cout << "adding " <<
zeroesToAdd / 8 << " empty eight bit blocks!\n";

//** add blocks of 8 bit length that will contains zero's
     for(int i = 0; i < zeroesToAdd / 8; i++)
          vectorBlockHoldingAsciiCharacters.push_back(0x00000000);

     //** we are finding ourself in 488 bits out 512 phase. Next
//** step is adding 1 in the binary representation in order to
//** form of eight bit blocks.
```

```
     bitset<64> theBig64BlobBit(lengthOfMessageInBits);
     if (displayDistanceFrom512Bit)
          cout << "l in a 64 bit binary blob: \n\t" <<
theBig64BlobBit << endl;

     //** divide the 64 bit big into 8 bit segments
     string big_64bit_string = theBig64BlobBit.to_string();

     //** take the first block and push it on the 56 position
     bitset<8> temp_string_holder1(big_64bit_string.substr(0,8));
     vectorBlockHoldingAsciiCharacters.
push_back(temp_string_holder1.to_ulong());

     //** take the rest of the blocks with 8 bits length and push
     for(int i = 8; i < 63; i=i+8)     {
          bitset<8>
temporaryStringHolder2(big_64bit_string.substr(i,8));

vectorBlockHoldingAsciiCharacters.
push_back(temporaryStringHolder2.to_ulong()); }

     //** just show in the console everything in order to know what
//** is happening in this freakin code
     if (displayResultsOfPadding)     {
        cout << "Current 512 bit pre-processed hash in binary: \n";
               for(int i = 0; i < vectorBlockHoldingAsciiCharacters.size();
               i=i+4)
                    cout << i << ": " << displayAsBinary(vectorBlockHolding
                    AsciiCharacters[i]) << "     "
                         << i + 1 << ": " << displayAsBinary(vectorBlock
                         HoldingAsciiCharacters[i+1]) << "     "
                         << i + 2 << ": " << displayAsBinary(vectorBlock
                         HoldingAsciiCharacters[i+2]) << "     "
                         << i + 3 << ": " << displayAsBinary(vectorBlock
                         HoldingAsciiCharacters[i+3]) << endl;

         cout << "Current 512 bit pre-processed hash in hex: \n";
          for(int i = 0; i < vectorBlockHoldingAsciiCharacters.size(); i=i+4)
```

```
                cout << i << ": " << "0x" + displayAsHex(vectorBlockHolding
                AsciiCharacters[i]) << "     "
                     << i + 1 << ": " << "0x" + displayAsHex(vectorBlock
                     HoldingAsciiCharacters[i+1]) << "     "
                     << i + 2 << ": " << "0x" + displayAsHex(vectorBlock
                     HoldingAsciiCharacters[i+2]) << "     "
                     << i + 3 << ": " << "0x" + displayAsHex(vectorBlock
                     HoldingAsciiCharacters[i+3]) << endl; }
     return vectorBlockHoldingAsciiCharacters; }

//** the goal of the function is to compute the hash of the message
string computingTheHash(const vector<unsigned long>
blockOf512BitPaddedMessage)
{
     //** the following words are from the NIST standard.
     unsigned long constantOf32BitWords[64] = {
     0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,
     0x923f82a4,0xab1c5ed5,0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,
     0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,0xe49b69c1,0xefbe4786,
     0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,
     0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,
     0x06ca6351,0x14292967,0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,
     0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85,0xa2bfe8a1,0xa81a664b,
     0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0x106aa070,
     0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,
     0x5b9cca4f,0x682e6ff3,0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,
     0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2 };

     //** the initial hash values
     unsigned long static InitialHashValueFor32Bit_0 = 0x6a09e667;
     unsigned long static InitialHashValueFor32Bit_1 = 0xbb67ae85;
     unsigned long static InitialHashValueFor32Bit_2 = 0x3c6ef372;
     unsigned long static InitialHashValueFor32Bit_3 = 0xa54ff53a;
     unsigned long static InitialHashValueFor32Bit_4 = 0x510e527f;
     unsigned long static InitialHashValueFor32Bit_5 = 0x9b05688c;
     unsigned long static InitialHashValueFor32Bit_6 = 0x1f83d9ab;
     unsigned long static InitialHashValueFor32Bit_7 = 0x5be0cd19;
```

```
     unsigned long Word[64];

     for(int t = 0; t <= 15; t++)     {
          Word[t] = blockOf512BitPaddedMessage[t] & 0xFFFFFFFF;

          if (displayWt)
               cout << "Word[" << t << "]: 0x" <<
displayAsHex(Word[t]) << endl; }

     for(int t = 16; t <= 63; t++) {
          Word[t] = SSIG1(Word[t-2]) +
Word[t-7] + SSIG0(Word[t-15]) + Word[t-16];

          Word[t] = Word[t] & 0xFFFFFFFF;

          if (displayWt)
               cout << "Word[" << t << "]: " << Word[t]; }

     unsigned long temporary_1;
     unsigned long temporary_2;
     unsigned long a = InitialHashValueFor32Bit_0;
     unsigned long b = InitialHashValueFor32Bit_1;
     unsigned long c = InitialHashValueFor32Bit_2;
     unsigned long d = InitialHashValueFor32Bit_3;
     unsigned long e = InitialHashValueFor32Bit_4;
     unsigned long f = InitialHashValueFor32Bit_5;
     unsigned long g = InitialHashValueFor32Bit_6;
     unsigned long h = InitialHashValueFor32Bit_7;

     if(displayWorkVariablesForT)
          cout << "         A        B        C        D        "
               << "E        F        G        H        T1   T2\n";

     for( int t = 0; t < 64; t++) {
          //** according to the NIST Standard and Specification,
//** the BSIG1 is incorrect. We will replace it with EP1
          temporary_1 = h + EP1(e) + CH(e,f,g) +
constantOf32BitWords[t] + Word[t];
```

```
          if ((t == 20) & displayT1Computation){
          cout << "h: 0x" << hex << h << "  dec:" << dec << h
                    << "  sign:" << dec << (int)h << endl;
          cout << "EP1(e): 0x" << hex << EP1(e) << "  dec:"
                << dec << EP1(e) << "  sign:" << dec << (int)EP1(e)
               << endl;
          cout << "CH(e,f,g): 0x" << hex << CH(e,f,g) << "  dec:"
                    << dec << CH(e,f,g) << "  sign:" << dec
                    << (int)CH(e,f,g) << endl;
          cout << "constantOf32BitWords[t]: 0x" << hex <<
constantOf32BitWords[t] << "  dec:" << dec
                    << constantOf32BitWords[t] << "  sign:" <<
     dec << (int)constantOf32BitWords[t] << endl;
          cout << "Word[t]: 0x" << hex << Word[t]
<< "  dec:" << dec << Word[t] << "  sign:" << dec
<< (int)Word[t] << endl;
          cout << "temporary_1 = 0x" << hex << temporary_1
<< "  dec:" << dec
               << temporary_1 << "  sign:" << dec <<
(int)temporary_1 << endl; }

          //** according to the NIST Standard and Specification,
//** the BSIG0 is incorrect. We will replace it with EP0
          temporary_2 = EP0(a) + MAJ(a,b,c);

          //** in order to get T2 we will display the variables
//** and operations
          if ((t == 20) & displayT2Computation) {
                  cout << "a: 0x" << hex << a << "  dec:" << dec << a
               << "  sign:" << dec << (int)a << endl;
             cout << "b: 0x" << hex << b << "  dec:" << dec << b
               << "  sign:" << dec << (int)b << endl;
             cout << "c: 0x" << hex << c << "  dec:" << dec << c
               << "  sign:" << dec << (int)c << endl;
             cout << "EP0(a): 0x" << hex << EP0(a) << "  dec:"
                << dec << EP0(a) << "  sign:" << dec << (int)EP0(a)
               << endl;
```

```
            cout << "MAJ(a,b,c): 0x" << hex
                << MAJ(a,b,c) << "  dec:"
              << dec << MAJ(a,b,c) << "  sign:" << dec
              << (int)MAJ(a,b,c) << endl;
   cout << "temporary_2 = 0x" << hex << temporary_2 << "  dec:" << dec <<
   temporary_2 << "  sign:" << dec << (int)temporary_2 << endl; }

          //** according to the NIST standard
          h = g;
          g = f;
          f = e;

//** Get the guarantee that we are still using 32 bits
          e = (d + temporary_1) & 0xFFFFFFFF;
          d = c;
          c = b;
          b = a;

//** Get the guarantee that we are still using 32 bits
          a = (temporary_1 + temporary_2) & 0xFFFFFFFF;

          //** display the content of each of the variable from
//** above according to the NIST standard.
      if (displayWorkVariablesForT)     {
             cout << "t= " << t << " ";
                 cout << displayAsHex (a) << " " << displayAsHex (b)
 << " " << displayAsHex (c) << " " << displayAsHex
(d) << " "   << displayAsHex (e) << " " << displayAsHex (f) << " " <<
displayAsHex (g) << " " << displayAsHex (h) << " " << endl; } }

//** display the content of each of the hash segment
     if(displayTheHashSegments) {
          cout << "InitialHashValueFor32Bit_0 = " << displayAsHex
(InitialHashValueFor32Bit_0) << " + " << displayAsHex (a) << " " <<
displayAsHex (InitialHashValueFor32Bit_0 + a) << endl;
```

```
          cout << "InitialHashValueFor32Bit_1 = " << displayAsHex
(InitialHashValueFor32Bit_1) << " + " <<
displayAsHex (b) << " " << displayAsHex
(InitialHashValueFor32Bit_1 + b) << endl;
          cout << "InitialHashValueFor32Bit_2 = " << displayAsHex
                   (InitialHashValueFor32Bit_2) << " + " <<
 displayAsHex (c) << " " << displayAsHex
 (InitialHashValueFor32Bit_2 + c) << endl;
          cout << "InitialHashValueFor32Bit_3 = " << displayAsHex
                    (InitialHashValueFor32Bit_3) << " + " <<
                    displayAsHex (d) << " " << displayAsHex
                    (InitialHashValueFor32Bit_3 + d) << endl;
          cout << "InitialHashValueFor32Bit_4 = " << displayAsHex
                    (InitialHashValueFor32Bit_4) << " + " <<
                    displayAsHex (e) << " " << displayAsHex
                    (InitialHashValueFor32Bit_4 + e) << endl;
          cout << "InitialHashValueFor32Bit_5 = " << displayAsHex
                    (InitialHashValueFor32Bit_5) << " + " <<
                    displayAsHex (f) << " " << displayAsHex
                    (InitialHashValueFor32Bit_5 + f) << endl;
          cout << "InitialHashValueFor32Bit_6 = " << displayAsHex
                 (InitialHashValueFor32Bit_6) << " + " <<
   displayAsHex (g) << " " << displayAsHex
   (InitialHashValueFor32Bit_6 + g) << endl;
cout << "InitialHashValueFor32Bit_7 = " << displayAsHex
  (InitialHashValueFor32Bit_7) << " + " << displayAsHex
  (h) << " " << displayAsHex (InitialHashValueFor32Bit_7
  + h) << endl;
     }

     //** for each hash add all the variables in order be sure that
//** we are still on the page with the 32 bit values
     InitialHashValueFor32Bit_0 = (InitialHashValueFor32Bit_0 + a)
& 0xFFFFFFFF;
```

```
     InitialHashValueFor32Bit_1 = (InitialHashValueFor32Bit_1 + b)
& 0xFFFFFFFF;
     InitialHashValueFor32Bit_2 = (InitialHashValueFor32Bit_2 + c)
& 0xFFFFFFFF;
     InitialHashValueFor32Bit_3 = (InitialHashValueFor32Bit_3 + d)
& 0xFFFFFFFF;
InitialHashValueFor32Bit_4 = (InitialHashValueFor32Bit_4 + e)
& 0xFFFFFFFF;
     InitialHashValueFor32Bit_5 = (InitialHashValueFor32Bit_5 + f)
& 0xFFFFFFFF;
     InitialHashValueFor32Bit_6 = (InitialHashValueFor32Bit_6 + g)
& 0xFFFFFFFF;
     InitialHashValueFor32Bit_7 = (InitialHashValueFor32Bit_7 + h)
& 0xFFFFFFFF;

     //** add the hash section in one piece one after the other in
//** order to obtain the 256 bit hash
     return displayAsHex(InitialHashValueFor32Bit_0) +
displayAsHex(InitialHashValueFor32Bit_1) + displayAsHex(InitialHashValue
For32Bit_2) +   displayAsHex(InitialHashValueFor32Bit_3) + displayAsHex(
InitialHashValueFor32Bit_4) + displayAsHex(InitialHashValueFor32Bit_5) +
                         displayAsHex(InitialHashValueFor32Bit_6) +
                        displayAsHex(InitialHashValueFor32Bit_7);
}
```

Hash functions are commonly used for digital signatures and in data integrity. A long message is generally hashed while dealing with digital signatures, and only the hash value is signed. The group that receives the message then hashes the message received and checks that the signature received is correct for this hash value. Below you can see a classification of the keyed cryptographic hash functions (see Table 2-2) and unkeyed cryptographic hash functions (see Table 2-3). Most of the functions are already implemented in C++ within the NIST Standard or other trusted resources, such as CrypTool[3].

[3]CrypTool, www.cryptool.org/en/

Table 2-2. *Keyed Cryptographic Hash Functions*

Name	Length of the tag	Type	References
BLAKE2	Arbitrary	Keyed hash function with prefix-MAC	[31][42]
BLAKE3	Arbitrary	Keyed hash function with supplied initializing vector (IV)	[32]
HMAC	-	-	[33]
KMAC	Arbitrary	Based on Keccak	[34][35]
MD6	512 bits	Merkle tree with NLFSR	[37]
PMAC	-	-	[38]
UMAC	-	-	[39]

Table 2-3. *Unkeyed Cryptographic Hash Functions*

Name	Length	Type	References
BLAKE-256	256 bits	HAIFA structure [41]	[40]
BLAKE-512	512 bits	HAIFA structure [41]	[40]
GOST	256 bits	Hash	[43]
MD2	128 bits	Hash	
MD4	128 bits	Hash	[44]
MD5	128 bits	Merkle-Damgard construction [36]	[45]
MD6	Up to 512 bits	Merkle-tree NLFSR	[37]
RIPEMD	128 bits	Hash	[46]
RIPEMD-128	128 bits	Hash	[46][47][48]
RIPEMD-256	-	Hash	
RIPEMD-160	160 bits	Hash	
RIPEMD-320	320 bits	Hash	
SHA-1	160 bits	Merkle-Damgard construction [36]	[61]

(*continued*)

Table 2-3. (*continued*)

Name	Length	Type	References
SHA-256	256 bits	Merkle-Damgard construction	[50][51][54]
SHA-384	384 bits		[52][54]
SHA-512	512 bits		[53][54]
SHA-224	224 bits	Merkle-Damgard construction	[55]
SHA-3 (Keccak)	Arbitrary	Sponge function [50]	[56][57]
Whirlpool	512 bits	Hash	[58][59][60]

Case Studies

Caesar Cipher Implementation in C++20

In this section, we will show a Caesar cipher implementation in C++20. The aim of this section is to explain how the above mentioned mathematical foundations can be useful during the implementation process and the advantages of understanding the basic mathematical mechanisms behind the algorithms. We will NOT dwell on the algorithm's mathematical history in this book. For any readers who want to go deep into the mathematical history, references [6-18] are recommended.

The encryption process used by a Caesar cipher can be represented as modular arithmetic by first transforming the letters into numbers. For this, we will follow *alphabet* $\mathcal{A} = \{A,\ldots,Z\} = 25$ in such way that $A = 0$, $B = 1$, …, $Z = 25$. The encryption of letter x is done by a shift n and mathematically can be described as

$$E_n(x) = (x + n)\ mod\ 26$$

The decryption is done in a similar way,

$$D_n(x) = (x - n)\ mod\ 26$$

Let's start the implementation of the algorithm (see Figure 2-9 and Listing 2-2).

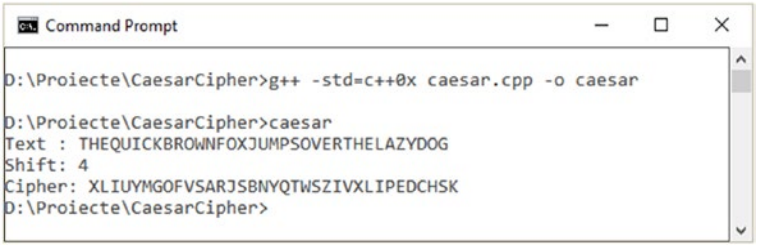


Figure 2-9. *The execution of a Caesar cipher*

The application is very simple and easy to interact with it.

Listing 2-2. Source Code for a Caesar Cipher Implementation

```
#include <iostream>
using namespace std;

// This function receives text and shift and
// returns the encrypted text
string encrypt(string text, int s)
{
    string result = "";

    // traverse text
    for (int i=0;i<text.length();i++)
    {
        // apply transformation to each character
        // Encrypt Uppercase letters
        if (isupper(text[i]))
            result += char(int(text[i]+s-65)%26 +65);

    // Encrypt Lowercase letters
    else
        result += char(int(text[i]+s-97)%26 +97);
    }

    // Return the resulting string
    return result;
}
```

```
// Driver program to test the above function
int main()
{
    string text="THEQUICKBROWNFOXJUMPSOVERTHELAZYDOG";
    int s = 4;
    cout << "Text : " << text;
    cout << "\nShift: " << s;
    cout << "\nCipher: " << encrypt(text, s);
    return 0;
}
```

Vigenére Cipher Implementation in C++20

The Vigenére cipher (see Figure 2-10 and Listing 2-3) is one of the classic methods of encrypting alphabetic text using a sequence of different Caesar ciphers based on keyword keys. You can see it in some of the documentations as a type of polyalphabetic substitution.

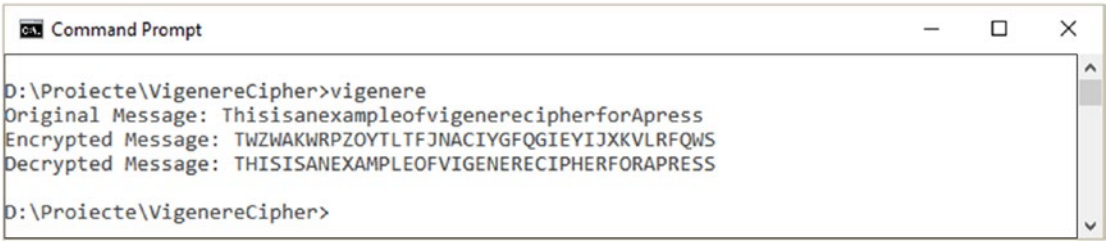


Figure 2-10. *Vigenére Cipher*

A short algebraic description of the cipher is as follows. The numbers are taken as numbers ($A = 0, B = 1, etc$) and an addition operation is performed as *modulo* 26. The Vigenére encryption E using K as the key can be written as

$$C_i = E_K\left(M_i\right) = \left(M_i + K_i\right) mod\ 26$$

and decryption D using the key K as

$$M_i = D_K\left(C_i\right) = \left(C_i - K_i\right) mod\ 26$$

in which $M = M_1 \ldots M_n$ is the message, $C = C_1 \ldots C_n$ represents the ciphertext, and $K = K_1 \ldots K_n$ represents the key obtained by repeating the keyword $[n/m]$ times in which m represents the keyword length.

Listing 2-3. The Source Code of a Vigenére Cipher

```
#include <iostream>
#include <string>
using namespace std;
class Vigenere {
   public:
       //** represents the key
      string key;

       //** the constructor of the class
       //** the chosen key
       Vigenere(string chosenKey) {
       for (int i = 0; i < chosenKey.size(); ++i) {
         if (chosenKey[i] >= 'A' && chosenKey[i] <= 'Z')
            this->key += chosenKey[i];
         else if (chosenKey[i] >= 'a' && chosenKey[i] <= 'z')
            this->key += chosenKey[i] + 'A' - 'a';
      }
   }
   string encrypt(string t)
   {
      string encryptedOutput;
      for (int i = 0, j = 0; i < t.length(); ++i) {
         char c = t[i];
         if (c >= 'a' && c <= 'z')
            c += 'A' - 'a';
         else if (c < 'A' || c > 'Z')
            continue;
         //** added 'A' to bring it in range
         //** of ASCII alphabet [ 65-90 | A-Z ]
```

```
            encryptedOutput += (c + key[j] - 2 * 'A') % 26 + 'A';
            j = (j + 1) % key.length();
         }
         return encryptedOutput;
      }
      string decrypt(string t) {
         string decryptedOutput;
         for (int i = 0, j = 0; i < t.length(); ++i) {
            char c = t[i];
            if (c >= 'a' && c <= 'z')
               c += 'A' - 'a';
            else if (c < 'A' || c > 'Z')
               continue;

            //** added 'A' to bring it in range of
            //** ASCII alphabet [65-90 | A-Z]
            decryptedOutput += (c - key[j] + 26) % 26 + 'A';
            j = (j + 1) % key.length();
         }
         return decryptedOutput;}};
int main() {
   Vigenere myVigenere("APRESS!WELCOME");
   string originalMessage
                  ="ThisisanexampleofvigenerecipherforApress";
   string enc = myVigenere.encrypt(originalMessage);
   string dec = myVigenere.decrypt(enc);
   cout << "Original Message: "<<originalMessage<< endl;
   cout << "Encrypted Message: " << enc << endl;
   cout << "Decrypted Message: " << dec << endl;
}
```

Conclusions

In this chapter, we gave a short introduction to the fundamentals of cryptographic primitives and mechanism. The chapter covered the following:

- Security and information security objectives
- The importance of the one-to-one, one-way, and trapdoor one-way functions in designing and implementing cryptographic functions
- Digital signatures and how they work
- Public-key cryptography and how it impacts developing applications
- Hash functions
- Case studies to illustrate the basic notions you need to know before advancing to high-level cryptographic concepts

The next chapter will go through the basics of probability theory, information theory, number theory, and finite fields. We will discuss their importance and how they are related during the implementation already existing in C++ and how they are useful for you as a developer.

References

[1] Simon Singh, *The Code Book: The Secrets Behind Codebreaking*, 2003.

[2] W. Diffie and M. Hellman, "New directions in cryptography" in *IEEE Trans. Information Theory*. 22, 6 (September 2006), 644–654. 2006. DOI: https://doi.org/10.1109/TIT.1976.1055638.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in *Communications ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[4] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms" in G.R. Blakley and D. Chaum (eds) *Advances in Cryptology. CRYPTO 1984. Lecture Notes in Computer Science, vol 196*. Springer, Berlin, Heidelberg.

[5] ISO/IEC 9796-2:2010 – Information Technology – Security Techniques – Digital Signature schemes giving message recovery. Available: https://www.iso.org/standard/54788.html.

[6] Bruce Schneier and Phil Sutherland, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition. ISBN: 978-0-471-12845-8. USA: John Wiley & Sons, Inc. 1995.

[7] William Stallings, *Cryptography and Network Security: Principles and Practice*. Upper Saddle River, N.J: Prentice Hall, 1999. Print.

[8] Douglas R. Stinson, *Cryptography: Theory and Practice*, first edition. ISBN: 978-0-8493-8521-6, CRC Press, Inc., USA. 1995.

[9] Neal Koblitz, *A Course in Number Theory and Cryptography*. New York: Springer-Verlag, 1994. Print.

[10] Neal Koblitz and A J. Menezes, *Algebraic Aspects of Cryptography*, 1999. Print.

[11] Oded Goldreich, *Foundations of Cryptography: Basic Tools*. Cambridge: Cambridge University Press, 2001. Print.

[12] Oded Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Berlin: Springer, 1999. Print.

[13] Michael G. Luby, *Pseudorandomness and Cryptographic Applications*. Princeton, NJ: Princeton University Press, 1996. Print.

[14] Bruce Schneier, *Secrets and Lies: Digital Security in a Networked World*. New York: John Wiley, 2000.

[15] Peter Thorsteinson and Arun Ganesh, *.NET Security and Cryptography*. Prentice Hall Professional Technical Reference, 2003.

[16] Adrian Atanasiu, *Criptografie (Cryptography) – Volume 1*. Publisher House: InfoData. ISBN: 978-973-1803-29-6, 978-973-1803-16-6. 2007. Available in Romanian Language.

[17] Adrian Atanasiu, *Protocoale de Securitate (Security Protocols) – Volume 2*. Publisher House: InfoData. ISBN: 978-973-1803-29-6, 978-973-1803-16-6. 2007. Available in Romanian Language.

[18] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot, *Handbook of Applied Cryptography* first edition. CRC Press, Inc., USA, ISBN: 978-0-8493-8523-0. 1996.

[19] Namespace System.Security.Cryptography. Available: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=netframework-4.8.

[20] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, second edition. Chapman & Hall/CRC. 2012.

[21] OpenPGP Library for .NET. Available online: www.didisoft.com/net-openpgp/

[22] Bouncy Castle .NET. Available online: www.bouncycastle.org/csharp/.

[23] Nethereum. Available from: https://github.com/Nethereum.

[24] Botan. Available online: https://botan.randombit.net/.

[25] Cryptlib. Available online: www.cs.auckland.ac.nz/~pgut001/cryptlib/.

[26] Crypto++. Available online: www.cryptopp.com/.

[27] Libgcrypt. Available online: https://gnupg.org/software/libgcrypt/.

[28] Libsodium. Available online: https://nacl.cr.yp.to/.

[29] Nettle. Available: www.lysator.liu.se/~nisse/nettle/.

[30] OpenSSL. Available: www.openssl.org/.

[31] J. Guo, P. Karpman, I. Nikolić, L. Wang, and S. Wu, "Analysis of BLAKE2. In: Benaloh J. (eds) Topics in Cryptology – CT-RSA 2014" in *Lecture Notes in Computer Science, vol 8366*. Springer, Cham. 2014.

[32] Blake3. Available online: https://github.com/BLAKE3-team/BLAKE3/.

[33] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: ekyed – Hashing for Message Authentication." RFC 2104, 1997.

[34] API KMAC. Available online: www.cryptosys.net/manapi/api_kmac.html.

[35] John Kelsey, Shu-jen Chang, Ray Perlner, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash" *in NIST Special Publication 800-185*. National Institute of Standards and Technology, December 2016.

[36] I.B. Damgard, "A design principle for hash functions" in *LNCS 435* (1990), pp. 516-527.

[37] Ronal L. Rivest, "The MD6 hash function. A proposal to NIST for SHA-3." Available online: http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf.

[38] PMAC. Available online: https://web.cs.ucdavis.edu/~rogaway/ocb/pmac.htm.

[39] UMAC. Available online: http://fastcrypto.org/umac/.

[40] BLAKE-256. Available online: https://docs.decred.org/research/blake-256-hash-function/.

[41] Eli Biham and Orr Dunkelman, "A Framework for Iterative Hash Functions - HAIFA. Second NIST Cryptographic Hash Workshop" via Cryptology ePrint Archive: Report 2007/278. August 24, 2006.

[42] BLAKE2 Official Implementation. Available online: https://github.com/BLAKE2/BLAKE2.

[43] GOST. Available online: https://tools.ietf.org/html/rfc5830.

[44] Roland L. Rivest, "The MD4 message digest algorithm," *LNCS, 537*, 1991, pp. 303-311.

[45] Roland L. Rivest, "The MD5 message digest algorithm," *RFC 1321*, 1992.

[46] RIPEMD-128. Available online: https://homes.esat.kuleuven.be/~bosselae/ripemd/rmd128.txt.

[47] RIPEMD-160. Available online: https://homes.esat.kuleuven.be/~bosselae/ripemd160.html.

[48] RIPEMD-160. Available online: https://ehash.iaik.tugraz.at/wiki/RIPEMD-160.

[49] The Sponge and Duplex Construction.Available online: https://keccak.team/sponge_duplex.html.

[50] Henri Gilbert and Helena Handschuh, "Security Analysis of SHA-256 and Sisters" in *Selected Areas in Cryptography 2003*: pp175–193.

[51] SHA256 .NET Class. Available online: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha256?view=netframework-4.8.

[52] SHA384 .NET Class. Available online: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha384?view=netframework-4.8.

[53] SHA512 .NET Class. Available online: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha512?view=netframework-4.8.

[54] Descriptions of SHA-256, SHA-384, and SHA-512. Available online: www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf.

[55] A 224-bit One-way Hash Function: SHA 224. Available online: www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf.

[56] Paul Hernandez, "NIST Releases SHA-3 Cryptographic Hash Standard." August 5, 2015.

[57] Morris J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". *Federal Inf. Process. STDS. (NIST FIPS) – 202*. August 4, 2015.

[58] Paulo S. L. M. Barreto, "The WHIRLPOOL Hash Function". Archived from the original on November 29, 2017. November 25, 2008. Retrieved August 9, 2018.

[59] Paulo S. L. M. Barreto and Vincent Rijmen, "The WHIRLPOOL Hashing Function". Archived from the original (ZIP) on October 26, 2017. May 24, 2003. Retrieved August 9, 2018.

[60] Whirlpool C# Implementation. Available online: `http://csharptest.net/browse/src/Library/Crypto/WhirlpoolManaged.cs`.

[61] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, "Finding Collisions in the Full SHA-1," *Crypto.* 2005.

CHAPTER 3

Mathematical Background and Its Applicability

This chapter will discuss the importance of the probability theory and its tools for modern cryptography. We will show how the elements and notions from the probability theory can be implemented in real-life applications and programs, and we will explain the most important steps for a professional cryptographer to follow in the implementation process of cryptographic algorithms.

The application of the probability theory to cryptography represents one of the challenging sides of cryptography and cryptanalysis. Between 1941 and 1942, Alan Turing (1912-1954) wrote a paper titled "The Applications of Probability to Cryptography"[1] (which was released by the Government Communications Headquarters (GCHQ) to the National Archives, HW/25/37[2]). This paper describes the application of the probability theory to code cracking. He started his paper with the Vigenère cipher. Turing provided proofs for the practical side by introducing and designing a unique method, the goal of which was to hide the entire complexity of the mathematical apparatus in cryptography, reducing the process down to a simple exercise using regular addition and a bit of trial and error. The tools introduced by him in the paper were *logarithms* and *probability*. To fully understand how the tools were applied, it was necessary to understand how the cipher worked.

The notions introduced in this chapter will help you to understand the basic mathematics in order to have a full appreciation of the solutions developed later.

[1]"The Applications of Probability to Cryptography," https://arxiv.org/abs/1505.04714

[2]Alan Turing Wartime Research Papers Released by GCHQ, https://discovery.nationalarchives.gov.uk/details/r/C11510465

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_3

We'll present each mathematical concept via the equations and mathematical expressions we will use during the implementation of the algorithms, providing examples of the implementations in C++. The implementations will be presented as case studies, counted from 1 to 10 (see Figures 3-1 through 3-12 and Listings 3-1 through 3-13).

Preliminaries

In this section, we will present the main concepts, giving the most appropriate definitions of *experiment, probability distribution, event, complementary event,* and *mutually exclusiveness.* The definitions are given in such way that you will find the intersection between theory and practice in a very fashionable and easy way to follow. The concepts described in this chapter will help you get a clear understanding and overview of the basic notions of what a cryptographic and cryptanalysis mechanism stands for and how it is projected using probabilities [1].

Definition 2.1 [1]. An *experiment* can be seen as a procedure that produces one of a mentioned set of outcomes. Each of the outcomes is individual. The ones that are possible are called simple events. The whole set formed out of the possible outcomes is known as a *sample space.*

In the following sections, we will discuss *discrete* sample spaces that have limited possible outcomes. We will write the simple events of a sample space as S labeled as s_1, s_2, …, s_n.

Definition 2.2 [1]. The probability distribution K over S is defined by a sequence of numbers, k_1, k_2, …, $k_n \geq 0$, and the sum of those numbers is equal to 1 ($k_1 + k_2 + \ldots + k_n = 1$). The number o_i can be interpreted as the *probability* of g_i. This is the outcome (result) of the processing experiment.

Definition 2.3 [1]. The *event* E represents a subset of the sample space S. In this situation, the *probability* that event E will occur, noted as $P(E)$, is defined as the sum of the probabilities o_i for all the simple events g_i which belong to E. If $g_i \in S$, $P(\{s_i\})$ is simply denoted as $P(s_i)$.

Definition 2.4 [1]. Let's consider E as an event. The *complementary event* is defined as being the set of simple events that don't belong to E, noted as $\overline{E}$.

Demonstration 2.1 [1]. If $E \subseteq S$ represents an event, the following should be considered:

- $0 \leq P(E) \leq 1$. In addition, $P(S) = 1$ and $P(\phi) = 0$, where ϕ represents an empty set.
- $P\left(\bar{E}\right) = 1 - P\left(E\right)$.
- If the results in S are just as likely, we can consider $P\left(E\right) = \frac{|E|}{|S|}$.

Definition 2.5 [1]. Consider E_1 and E_2 two *mutually exclusive* events. They are mutually exclusive if $P(E_1 \bigcap E_2) = 0$. The showing nature of one or two events will have the chance to exclude the case that the other has the possibility of taking place.

Definition 2.6 [1]. Take as an example two events, E_1 and E_2.

- $P(E_1) \leq P(E_2)$ will be if $E_1 \subseteq P(E_2)$.
- $P(E_1 \cup E_2) + P(E_1 \cap E_2) = P(E_1) + P(E_2)$. Accordingly, if E_1 and E_2 are considered mutually exclusive, then the following expression takes place: $P(E_1 \cup E_2) = P(E_1) + P(E_2)$.

Conditional Probability

Definition 2.7 [1]. Let's consider E_1 and E_2 as being two events, with $P(E_2) > 0$. The *conditional probability* for E_1 to give E_2 is written as $P(E_1|E_2)$ and it is expressed as

$$P\left(E_1|E_2\right) = \frac{P\left(E_1 \cap E_2\right)}{P\left(E_2\right)}.$$

$P(E_1|E_2)$ measures the probability of how event E_1 will take place, given that E_2 has occurred.

Definition 2.8 [1]. Consider E_1 and E_2 as two events. Their relationship is one of *independency* if $P(E1 \cap E_2) = P(E_1)P(E_2)$.

Definition 2.9 *(Bayes' Theorem)* [1]. Assuming that we have two events, E_1 and E_2, with $P(E_2) > 0$, then

$$P\left(E_1|E_2\right) = \frac{P\left(E_1\right)P\left(E_2|E_1\right)}{P\left(E_2\right)}.$$

Random Variables

Let's take into consideration a sample space S that has the distribution probability of P.

Definition 2.10 [1]. Let X be a *random variable.* Declare a function that is applied on S for the set of real numbers. For each event $s_i \in S$, X, there will be a real number assigned $X(s_i)$.

Definition 2.11 [1]. Let X be the random variable on S. The *mean* or *expected value* of X is defined as follows:

$$E(X) = \sum_{s_i \in S} X(s_i) P(s_i).$$

For the C++ implementation of a mean or expected value, refer to Case Study 3: Computing the Mean of Probability Distribution.

Demonstration 2.12 [1]. Consider X to be a random variable on S. In this case, we have the following expression:

$$E(X) = \sum_{x \in \mathbb{R}} x \cdot P(X = x).$$

Demonstration 2.13 [1]. Let's consider the following random variables of S: X_1, X_2, ..., X_m. The following are real numbers: a_1, a_2, ..., a_m. Then we have the following expression to be satisfied:

$$E\left(\sum_{i=1}^{m} a_i X_i\right) = \sum_{i=1}^{m} a_i E(X_i).$$

Definition 2.14. Let's consider X as a random variable. The *variance* of X of means μ is defined by the non-negative number that is expressed by

$$Var(X) = E\left((X - \mu)^2\right).$$

For the C++ implementation of the mean or expected value, refer to Case Study 4: Computing the Variance.

The *standard deviation* of X is defined by the non-negative square root of $Var(X)$.

For the C++ implementation of the mean or expected value, refer to Case Study 5: Computing the Standard Deviation.

Birthday Problem

Definition 2.15 [1]. Consider two positive integers a, b with $a \geq b$, where the number $m^{(n)}$ is defined as follows:

$$m^{(n)} = m(m-1)(m-2)\ldots(m-n+1).$$

Definition 2.15 [1]. Consider two non-negative integers a, b with $a \geq b$. The *Stirling number of the second kind*, represented and noted as $\left\{ {a \atop b} \right\}$, is expressed as follows:

$$\left\{ {a \atop b} \right\} = \frac{1}{b!}\sum_{i=0}^{n}(-1)^{b-i}\binom{b}{i}i^{a}.$$

The case of $\left\{ {0 \atop 0} \right\} = 1$ is considered an exception.

Demonstration 2.16 [1]. As an example, let's consider the classic occupancy problem by illustrating it with an urn that contains a balls. The balls are numbered (or labeled) from 1 to m. Let's image a scenario in which b balls are extracted from the urn one at a time and replaced in the same time, and with their numbers listed. The chance (probability) for l different balls to have been drawn is

$$P_1(a,b,l) = \left\{ {b \atop l} \right\}\frac{a^{(l)}}{a^{b}}, 1 \leq l \leq b.$$

The birthday problem represents a special case of the occupancy problem.

Demonstration 2.17 [1]. We take into consideration the *birthday problem*, where we have an jar with a balls that are numbered from 1 to a. Assume that a specific number of balls, h, are extracted from the urn one at a time and replaced, with their numbers listed.

Case 2.17.1 [1]. The probability of at least one coincidence, for example a ball that is drawn at least twice from the urn, is

$$P_2(a,h) = 1 - P_1(a,h,h) = 1 - \frac{a^{(h)}}{a^{h}}, 1 \leq h \leq m.$$

Case 2.17.2 [1]. Let's consider h the number of balls extracted from the jar. If $h = O\left(\sqrt{a}\right)$ and $a \to \infty$, then the following expression will take place:

$$P_2(a,h) \to 1 - \exp\left(-\frac{h(h-1)}{2a} + O\left(\frac{1}{\sqrt{a}}\right)\right) \approx 1 - \exp\left(-\frac{h^2}{2a}\right).$$

The demonstration that we provided explains why the probability distribution is known as the *birthday surprise* or *birthday paradox*. The probability that at least 2 people in a room of 23 people have the same birthday is $P_2(365, 23) \approx 0.507$, which is surprisingly large. The quantity $P_2(365, h)$ increases as h increases. As an example, $P_2(365, 30) \approx 0.706$.

For the C++ implementation of the birthday paradox, refer to Case Study 4: Birthday Paradox.

Information Theory

Entropy

Let's denote with X a random variable that takes on a finite set of values $x_1, x_2, \ldots, x_n$, with the probability $P(X = x_i) = p_i$, where $0 \le p_i \le 1$ for each i, $1 \le i \le n$, in which the following sum expression take place:

$$\sum_{i=1}^{n} p_i = 1.$$

As well, let's declare Y and Z random variables, which will take a finite set of values [1].

The entropy of A is defined as a mathematical measure that is characterized as the amount of information that is provided by observation o.

Definition 2.18 [1]. Let's denote A as a random variable, so the *entropy* or uncertainty of A is defined by the expression

$$H(A) = -\sum_{j=1}^{m} p_j \lg p_j = \sum_{j=1}^{m} p_j lg\left(\frac{1}{p_j}\right)$$

where, through convention,

$$p_i \cdot lgp_i = p_i \cdot lg\left(\frac{1}{p_i}\right) = 0 \text{, if } p_i = 0.$$

Definition 2.19 [1][5]. Let's consider A and B, two random variables. The *joint entropy* is defined by expression

$$H(A,B) = \sum_{a,b} P(A = a, B = b) \lg\left(P(A = a, B = b)\right),$$

where a and b go through all of the values within the random variables, A and B.

Definition 2.20 [1]. Let's consider two random variables A and B, and suppose that the *conditional entropy* of A given $B = m$ is expressed as

$$H\left(A|B=v\right)=-\sum_{m}P\left(A=m|B=v\right)\lg\left(P\left(A=m|B=v\right)\right),$$

where m goes through all of the values within the random variable A. In this case, the *conditional entropy* of A given B, called also the *equivocation* of B about A, is declared as

$$H\left(A|B\right)=\sum_{m}P\left(B=m\right)H\left(A|B=m\right),$$

where m (which is an index) goes through all of the values of B.

Number Theory

Integers

Starting from the idea that a set of integers $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$ is represented by the symbol $\mathbb{Z}$, the following definitions will occur.

Definition 2.21 [1]. Let's assume that we have two integers, x and y. We will start from the idea that x *divides* y if there exists an integer d in such way that $y = x \cdot d$. If x is dividing y, then we can state that $x \mid y$.

Definition 2.22 (Division algorithm for integers) [1]. Consider two integers, x and y with $y \geq 1$. An ordinary long division of x by y holds the integers *quot* (*quotient*) and *rem* (*remainder*) in such way that

$$x = qout \cdot y + rem, \text{ where } 0 \leq rem < y.$$

Definition 2.23 [1]. Consider d as integer. Note that the *common divisor* of x and y exists if $d \mid x$ and $d \mid y$.

Definition 2.24 [1]. Assume that we have a non-negative integer e. The non-negative integer e is known as being the *greatest common divisor (gcd)* of the integers x and y. We note it as $e = \gcd(x, y)$, if

a. e is a common divisor x and y;

b. $d \mid x$ and $d \mid y$, then $d \mid e$.

Definition 2.25 [1]. Assume a non-negative integer e. The non-negative integer e is known as being the *least common multiple (lcm)* of integers x and y. We note it as $e = lcm(x, y)$, if

a. $x \mid e$ and $y \mid e$;

b. $x \mid d$ and $x \mid d$, then $e \mid d$.

Algorithms in $\mathbb{Z}$

Let's consider two non-negative integers, a and b, with $a \leq n$. Note that the number of bits from the binary representation of n is represented as $\lfloor lg\, n \rfloor + 1$. This value will be approximated by lg$n$. The bit operations related to the four basic operations for integers using the classical algorithms are shown in Table 3-1.

Table 3-1. *The Bit Complexity of the Basic Operation in Z*

Operation		Bit Complexity
Addition	$a + b$	$O(lga + lgb) = O(lgn)$
Subtraction	$a - b$	$O(lga + lgb) = O(lgn)$
Multiplication	$a \cdot b$	$O((lga)(lgb)) = O((lgn)^2)$
Division	$a = q \cdot b + r$	$O((lgq)(lgb)) = O((lgn)^2)$

Demonstration 2.26 [1]. The integers a and b are positive numbers with $a > b$, so $\gcd(a, b) = \gcd(b, a \bmod b)$.

Algorithm 2.27 [1]. The Euclidean algorithm for computing the *gcd* for two integers

$$INPUT: \quad a \text{ and } b, \text{two non} - \text{negative integers with respect for } a \geq b$$

$$OUTPUT: \quad \text{the gcd}$$

$$1.\ While\ b \neq 0\ then$$

$$1.1.\ Set\ r \leftarrow a \bmod b,\ a \leftarrow b, b \leftarrow r.$$

$$2.\ Return(a).$$

The Euclidean algorithm can be extended so that it will not only yield the gcd of two integers *a* and *b*, but also integers *x* and *y*, which will satisfy $ax + by = d$.

Algorithm 2.28 [1]. Pseudocode for extended Euclidean algorithm

$$INPUT: \quad x \text{ and } y, \text{non} - \text{negative numbers with the following condition } a \geq b$$

$$OUTPUT: \ h = \gcd(x,y) \text{ and integers } w, z \text{ which satisfies } xw + yz = h$$

$$1. \text{ If } y = 0, \text{then}$$

$$h \leftarrow x$$

$$w \leftarrow 1$$

$$z \leftarrow 0$$

$$return(h,w,z).$$

$$2. \text{ Declare and initialize } w_2 \leftarrow 1, w_1 \leftarrow 0, z_2 \leftarrow 0, z_1 \leftarrow 1.$$

$$3. \text{ While } y > 0, \text{then}$$

$$3.1. \ quotient \leftarrow \frac{x}{y}$$

$$remainder \leftarrow x - quotient \cdot y;$$

$$w \leftarrow w_2 - quotient \cdot w_1; z \leftarrow z_2 - quotient \cdot z_1.$$

$$3.2. \ x \leftarrow y$$

$$y \leftarrow remainder$$

$$w_2 \leftarrow w_1$$

$$w_1 \leftarrow w$$

$$z_2 \leftarrow z_1$$

$$z_1 \leftarrow z.$$

$$4. \text{ Set } h \leftarrow x, w \leftarrow w_2, z \leftarrow z_2.$$

$$return(h,w,z).$$

In Case Study 7: (Extended) Euclidean Algorithm, we provide an implementation using C++ for both types of the algorithm, Euclidean and extended Euclidean algorithm.

The Integer Modulo *n*

Consider p a positive integer.

Definition 2.30 [1]. Let i and j be two integers. We allege that g *is congruent to j modulo q*. The notation used is

$$i \equiv j \,(mod\ q), \text{ if } q \text{ will divide } (i - j).$$

So q is called the *modulus* of the congruence.

Definition 2.31 [1]. Consider $n \in Z_q$. The *multiplicative inverse* of n modulo q is represented by an integer $x \in \mathbb{Z}_q$ in such way that $n\,x \equiv 1\ (mod\ q)$. If there is an n that exists, then that n is unique, and we state that n is *invertible*, or a *unit.* The inverse of n is noted as n^{-1}.

Case Study 8: Computing the Multiplicative Inverse under Modulo m provides a C++ implementation of a multiplicative inverse under modulo q.

Definition 2.32. Chine Remainder Theorem (CRT) [1]. The integers $n_1, n_2, \ldots, n_k$, represents a pairwise (occurring in pairs) that is relatively prime. Let's consider the following system formed out of simultaneous congruences

$$\begin{aligned} j &\equiv v_1 \quad (mod\ g_1) \\ j &\equiv v_2 \quad (mod\ g_2) \\ &\vdots \\ l &\equiv v_n \quad (mod\ g_k) \end{aligned}$$

as a system that has a unique solution modulo $g = g_1\, g_2 \cdots g_k$.

Case Study 9: Chinese Remainder Theorem provides a C++ implementation of the Chinese Remainder Theorem.

Definition 2.33. Gauss's Algorithm [1]. As you saw in the Chinese Remainder Theorem, the solution y for concurrent congruences may be calculated as $y = \sum_{h=1}^{l} b_h \cdot R_h \cdot L_h \bmod q$, where $R_i = q/q_h$ and $L_h = R_h^{-1} \bmod q_i$. The listed operations can be done in $O((lgq)^2)$ bit operations.

Algorithms $\mathbb{Z}_m$

Consider a positive integer m. As you have seen, the addition of the elements of $\mathbb{Z}_m$ is defined as

$$(x+y) \bmod m = \begin{cases} x+y, & if\ x+y<m, \\ x+y-m, & if\ x+y \geq m \end{cases}.$$

Algorithm 2.34 [1]. Pseudocode for computing the multiplicative inverses in $\mathbb{Z}m$

$INPUT:$ $x \in \mathbb{Z}_m$

$OUTPUT:$ $x^{-1} \bmod m$

1. *Use the extended Euclidean algorithm and find the integers w and z such that* $xw+nz=h$, *where* $h=\gcd(x,n)$

2. *If* $h>1$, *we will have* $x^{-1} \bmod q$ *which will not exist. Else, return* (w).

Algorithm 2.35 [1]. Repeated square-and-multiply algorithm for exponentiation in $\mathbb{Z}_m$

$INPUT:$ $x \in \mathbb{Z}_m$, *and integer* $0 \leq t < m$ *whose binary representation is* $t=\sum_{j=0}^{o} t_j 2^j$.

$OUTPUT:$ $x^l \bmod m$

1. *Set* $y \leftarrow 1$. *If* $t=0$, *then return*(y).
2. *Set* $C \leftarrow x$.
3. *If* $t_0=1$, *then set* $y \leftarrow x$.
4. *For j from* 1 *to k, do*
   4.1 *Set* $C \leftarrow C^2 \bmod m$.
   4.2 *If* $t_j=1$, *then set* $y \leftarrow C \cdot y \bmod m$.
5. *Return*(y).

The Legendre and Jacobi Symbols

In order to check if an integer is a quadratic residue in a specific modulo, the Legendre symbol is the perfect tool for this purpose.

Definition 2.36 [1]. Consider q an odd prime and x an integer. The *Legendre symbol,* noted as $\left(\frac{x}{q}\right)$ is defined as

$$\left(\frac{x}{q}\right)=\begin{cases}0, & if\ q|x\\ 1, & if\ x\in W_q\\ -1, & if\ x\in \overline{W_q}\end{cases}.$$

Properties 2.37. Properties of the Legendre Symbol [1]. The following properties will be considered. The following properties are known as the properties of the Legendre Symbol. For the following properties, consider m to be an odd prime. Let's declare two integers $x, y \in Z$. The next properties specific to the Legendre symbol are listed as

1. $\left(\frac{x}{m}\right)\equiv x^{\frac{m-1}{2}}\ (mod\ m)$. In the particular case, $\left(\frac{1}{m}\right)=1$ and $\left(\frac{-1}{m}\right)=(-1)^{\frac{m-1}{2}}$. Since $-1\in W_m$ if $m\equiv 1\ (mod\ 4)$ and $-1\in\overline{W_m}$ if $m\equiv 3\ (mod\ 4)$.
2. $\left(\frac{xy}{m}\right)=\left(\frac{x}{m}\right)\left(\frac{y}{m}\right)$. Since if $x\in\mathbb{Z}_q^*$, then $\left(\frac{x^2}{m}\right)=1$.
3. If $x\equiv y\ (mod\ m)$, then $\left(\frac{x}{m}\right)=\left(\frac{y}{m}\right)$.
4. $\left(\frac{2}{m}\right)=(-1)^{\frac{(m^2-1)}{8}}$. Since $\left(\frac{2}{m}\right)=1$ if $m\equiv 1$ or $7\ (mod\ 8)$, and $\left(\frac{2}{m}\right)=-1$ if $m\equiv 3$ or $5\ (mod\ 8)$.
5. If m represent an odd prime distinct from p, we have

$$\left(\frac{t}{m}\right)=\left(\frac{m}{t}\right)(-1)^{\frac{(t-1)(m-1)}{4}}.$$

The Jacobi symbol represents a generalization of the Legendre Symbol for integers n that are not odd and are also not necessarily prime.

Definition 2.38. Jacobi Definition [1]. Let $m \geq 3$ represent an odd with a prime factorization as

$$m = v_1^{h_1} v_2^{h_2} \ldots v_j^{h_j}.$$

The Jacobi symbol $\left(\frac{x}{m}\right)$ has the following expression:

$$\left(\frac{x}{m}\right) = \left(\frac{x}{m_1}\right)^{h_1} \left(\frac{x}{m_2}\right)^{h_2} \ldots \left(\frac{x}{m_j}\right)^{h_j}.$$

We need to take into consideration the fact that if n is prime, the Jacobi symbol will be a Legendre symbol.

Properties 2.39. Jacobi Symbol Properties [1]. Consider $x \geq 3$ and $y \geq 3$ to be odd integers, and $i, j \in \mathbb{Z}$. The Jacobi symbol will have the following properties:

1. $\left(\frac{i}{y}\right) = 0, 1, or -1$. More than this, $\left(\frac{i}{y}\right) = 0$ if and only if $\gcd(i, y) \neq 1$.
2. $\left(\frac{ij}{y}\right) = \left(\frac{i}{y}\right)\left(\frac{j}{y}\right)$. Hence if $i \in \mathbb{Z}_m^*$, then $\left(\frac{i}{y}\right) = 1$.
3. $\left(\frac{i}{yx}\right) = \left(\frac{i}{y}\right)\left(\frac{i}{x}\right)$.
4. If $i \equiv j \,(mod\, y)$, then $\left(\frac{i}{y}\right) = \left(\frac{j}{y}\right)$.
5. $\left(\frac{1}{y}\right) = 1$.
6. $\left(\frac{-1}{y}\right) = (-1)^{\frac{(y-1)}{2}}$. Hence $\left(\frac{-1}{y}\right) = 1$ if $y \equiv 1 \,(mod\, 4)$, and $\left(\frac{-1}{y}\right) = -1$ if $y \equiv (3 \,mod\, 4)$.
7. $\left(\frac{2}{y}\right) = (-1)^{\frac{y^2-1}{8}}$. Hence $\left(\frac{2}{y}\right) = 1$ if $y \equiv 1$ or $7 \,(mod\, 8)$, and $\left(\frac{2}{y}\right) = -1$ if $y \equiv 3$ or $5 \,(mod\, 8)$.
8. $\left(\frac{x}{y}\right) = \left(\frac{y}{x}\right)(-1)^{\frac{(x-1)(y-1)}{4}}$. In other words, $\left(\frac{x}{y}\right) = \left(\frac{y}{x}\right)$ unless both x and y are congruent to 3 modulo 4, in which case $\left(\frac{x}{y}\right) = -\left(\frac{y}{x}\right)$.

Algorithm 2.40. Pseudocode of Jacobi Symbol. Pseudocode for Legendre symbol [1]

$$JACOBI\left(h,k\right)$$

$$INPUT: \quad Odd\ integer\ k \geq 3\ and\ an\ integer\ h, 0 \leq h < k$$

$$OUTPUT: \quad The\ Jacobi\ symbol\left(\frac{h}{k}\right)$$

1. *If* $h = 0$, *then return* 0.
2. *If* $h = 1$, *then return* 1.
3. *Write* $h = 2^t h_1$, *where* h_1 *is odd.*
4. *If* t *is even, then set* $g \leftarrow 1$. *Else set* $g \leftarrow 1$ *if* $k \equiv 1$ *or* $7\ (mod\ 8)$, *or set* $g \leftarrow -1$ *if* $k \equiv 3$ *or* $5\ (mod\ 8)$.
5. *If* $k \equiv 3\ (mod\ 4)$ *and* $h_1 \equiv 3\ (mod\ 4)$, *then set* $g \leftarrow -g$.
6. *Set* $k_1 \leftarrow k\ mod\ h_1$.
7. *If* $h_1 = 1$, *then return*(g); *else return* $(g \cdot JACOBI(k_1, h_1))$.

Finite Fields

Basic Notions

Definition 2.41 [1]. Consider F to be a *finite field* that contains a finite number of elements. The *order of* F represents the number of elements in F.

Definition 2.42 [1]. The finite fields are characterized through a special uniqueness.

1. Let's assume if P represents a finite field, then P will contain h^j elements for a prime h and integer $j \geq 1$.
2. For each prime power order h^j, we have a unique finite field of order h^j. The field is noted as $\mathbb{G}_{h_j}$ or in some other literature references as $GF(h^j)$.

Definition 2.43 [1][5]. If G_h represents a finite field of order $h = a^m$, a is prime, then the characteristic of $\mathbb{F}_h$ is p. More than this, h has a copy of $\mathbb{Z}_a$ as a subfield. Since $\mathbb{F}_h$ can be viewed as an extension field of $\mathbb{Z}_a$ of degree m.

Polynomials and the Euclidean Algorithm

The below two algorithms represent the foundation for understanding how to compute the *gcd* for two polynomials, $g(x)$ and $h(x)$, both being in $\mathbb{Z}_p[x]$.

Algorithm 2.43. Euclidean Algorithm for $\mathbb{Z}p[x]$ [1]

$$INPUT: \quad Two\ polynomials\ g(x), h(x) \in \mathbb{Z}_p[x]$$

$$OUTPUT: \quad \gcd of\ g(x)\ and\ h(x)$$

1. *While* $h(x) \neq 0$, *then*

   set $r(x) \leftarrow g(x)\ mod\ h(x)$, $g(x) \leftarrow h(x)$, $h(x) \leftarrow r(x)$.

2. *Return* $g(x)$.

Algorithm 2.43. Extended Euclidean Algorithm for $\mathbb{Z}p[x]$ [1]

$$INPUT: \quad Two\ polynomials\ g(x), h(x) \in \mathbb{Z}_p[x]$$

$OUTPUT:$ $d(x) = \gcd(g(x), h(x))$ *and polynomials* $s(x), t(x) \in \mathbb{Z}_p[x]$, *which will satisfy* $s(x)g(x) + t(x)h(x) = d(x)$.

1. If $h(x) = 0$, *then set* $d(x) \leftarrow g(x)$, $s(x) \leftarrow 1$, $t(x) \leftarrow 0$

   return $(d(x), s(x), t(x))$.

2. Set $s_2(x) \leftarrow 1$, $s_1(x) \leftarrow 0$, $t_2(x) \leftarrow 0$, $t_1(x) \leftarrow 1$.

3. While $h(x) \neq 0$, *then*

   a. $g(x) \leftarrow g(x)\ div\ h(x)$, $r(x) \leftarrow g(x) - h(x)q(x)$

   b. $s(x) \leftarrow s_2(x) - q(x)s_1(x)$, $t(x) \leftarrow t_2(x) - q(x)t_1(x)$

   c. $g(x) \leftarrow h(x)$, $h(x) \leftarrow r(x)$

   d. $s_{2(x)} \leftarrow s_1(x)$, $s_1(x) \leftarrow s(x)$, $t_2(x) \leftarrow t_1(x)$, *and* $t_1(x) \leftarrow t(x)$.

4. Set $d(x) \leftarrow g(x)$, $s(x) \leftarrow s_2(x)$, $t(x) \leftarrow t_2(x)$.

5. Return $d(x)$, $s(x)$, $t(x)$..

Case Study 1: Computing the Probability of an Event Taking Place

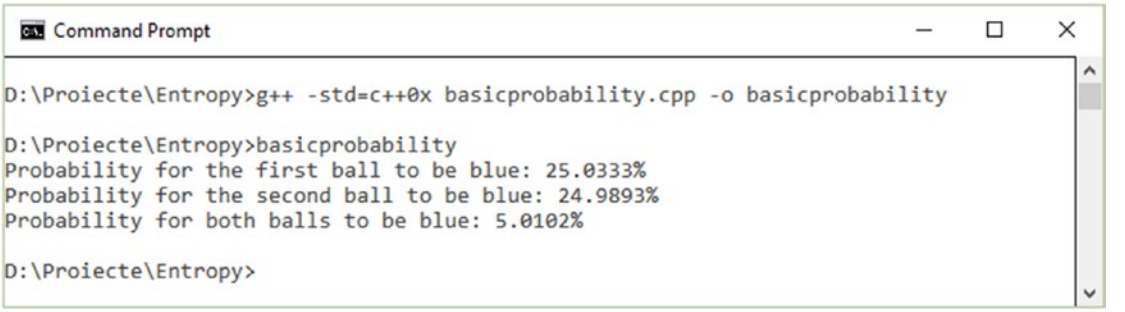


Figure 3-1. *Output for computing the probability*

Listing 3-1. Source Code

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>

enum ColorTypes {
      Blue,
      NotBlue } ;

//** create a sequence container
typedef std::vector<ColorTypes> backpack;

backpack initializeBackpack(unsigned blue_balls, unsigned
                            differentBalls)
{
    backpack backpackOfBalls ;

    for (unsigned i=0; i<blue_balls; ++i)
         backpackOfBalls.emplace_back(Blue);

    for (unsigned i=0; i<differentBalls; ++i)
         backpackOfBalls.emplace_back(NotBlue);

    return backpackOfBalls; }
```

```
void randomize(backpack & backpackOfBalls) {
      //** Mersenne Twister - pseudo-random generator
      //** on 32-bit number using the state size of 19937 bits/
      //** std:random_device() will help us to generate a
      //** non-deterministic random numbers
     static std::mt19937 engine((std::random_device()()));
            //** we will rearrange the elements in the
      //** following range [first, second] as follows fist =
      //** backpackOfBalls.begin() and second =
      //** backpackOfBalls.end()
      //** using "engine" declared above as a uniform random
      //** number generator
     std::shuffle(backpackOfBalls.begin(),
                  backpackOfBalls.end(), engine);
}

int main()
{
      //** constants initializations
      const unsigned theTotalOfSamples = 1000000;
      const unsigned blue_balls = 4;
      const unsigned differentBalls = 12;

      unsigned theFirstIsBlue = 0;
      unsigned bothAreBlue = 0;
      unsigned theSecondIsBlue = 0;

      auto backpackOfBalls = initializeBackpack(blue_balls,
                                                differentBalls) ;

      for (unsigned i=0; i<theTotalOfSamples; ++i)
      {
        randomize(backpackOfBalls);

        if (backpackOfBalls[0] == Blue)
            ++theFirstIsBlue;

        if (backpackOfBalls[1] == Blue)
            ++theSecondIsBlue;
```

```
        if (backpackOfBalls[0]==Blue&&backpackOfBalls[1]==Blue)
            ++bothAreBlue;
    }

    float probabilityOfFirstBallToBeBlue =
                    static_cast<float>(theFirstIsBlue) /
                                        theTotalOfSamples;

    float probabilityForBothBallsToBeBlue =
                              static_cast<float>(bothAreBlue) /
                                            theTotalOfSamples;

    float probabilityForSecondBallToBeRed =
                            static_cast<float>(theSecondIsBlue) /
                                        theTotalOfSamples;

    std::cout << "Probability for the first ball to be blue: "
          << probabilityOfFirstBallToBeBlue * 100.0 << "%\n" ;

    std::cout<< "Probability for the second ball to be blue: "
         << probabilityForSecondBallToBeRed * 100.0 << "%\n" ;

    std::cout << "Probability for both balls to be blue: "
         << probabilityForBothBallsToBeBlue * 100.0 << "%\n" ;
}
```

Case Study 2: Computing the Probability Distribution

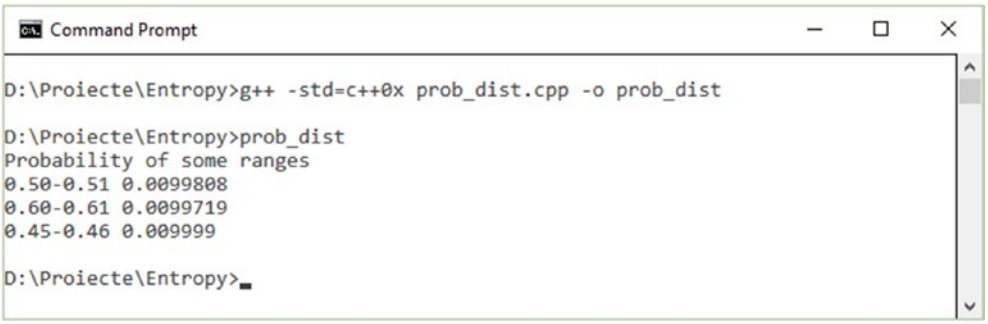


Figure 3-2. *Output of the probability distribution*

Listing 3-2. Source Code

```
//** this will be used for computing the distribution
#include <random>
#include <iostream>

using namespace std;


int main()  {
    //** declare default_random_engine object
    //** we will use it as a random number
    //** we will provide a seed for default_random_engine
    //** if a pseudo random is necessary
    default_random_engine gen;
    double x=0.0, y=1.0;

   //** initialization of the probability distribution
    uniform_real_distribution<double> dist(x, y);

    //** the number of experiments
    const int numberOfExperiments = 10000000;

    //** the number of ranges
    const int numberOfRanges = 100;
    int probability[numberOfRanges] = {};
    for (int k = 0; k < numberOfExperiments; ++k) {
        // using operator() function
        // to give random values
        double no = dist(gen);
        ++probability[int(no * numberOfRanges)]; }

    cout << "Probability of some ranges" << endl;
    //** show the probability distribution of some ranges
    //** after 1000 times values are generated
    cout << "0.50-0.51"<<" "<<
      (float)probability[50]/(float)numberOfExperiments<<endl;
    cout << "0.60-0.61"<<" "<<
      (float)probability[60]/(float)numberOfExperiments<<endl;
```

```
    cout << "0.45-0.46"<<" "<<
      (float)probability[45]/(float)numberOfExperiments<<endl;

    return 0;
}
```

Case Study 3: Computing the Mean of the Probability Distribution

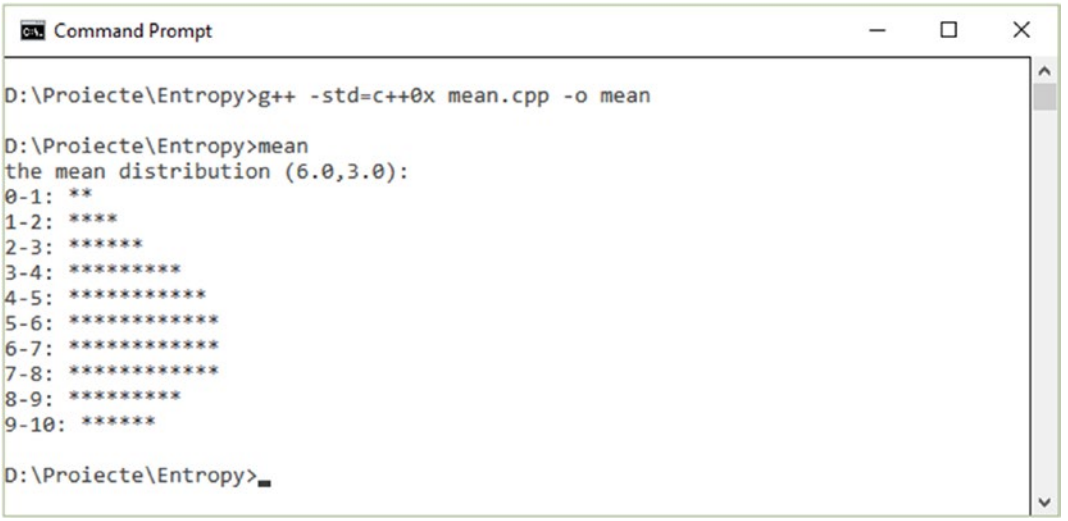


Figure 3-3. *Output for the mean of theprobability distribution*

Listing 3-3. Source Code

```
#include <iostream>
#include <string>
#include <random>

int main()
{
  //** the constant represents the number of experiments
  const int numberOfExperiments=10000;
  //** the constant represents the
```

```
  //** maximum number of stars to distribute
  const int numberOfStarsToDistribute=100;

  std::default_random_engine g;
  std::normal_distribution<double> dist(6.0,3.0);

  int prob[10]={};

  for (int k=0; k<numberOfExperiments; ++k) {
    double no = dist(g);
    if ((no>=0.0)&&(no<10.0)) ++prob[int(no)];
  }

  std::cout << "the mean distribution (6.0,3.0):" << std::endl;

  for (int l=0; l<10; ++l) {
    std::cout << l << "-" << (l+1) << ": ";
    std::cout <<
           std::string(prob[l]*numberOfStarsToDistribute/
numberOfExperiments,'*') << std::endl;
  }

  return 0;
}
```

Case Study 4: Computing the Variance

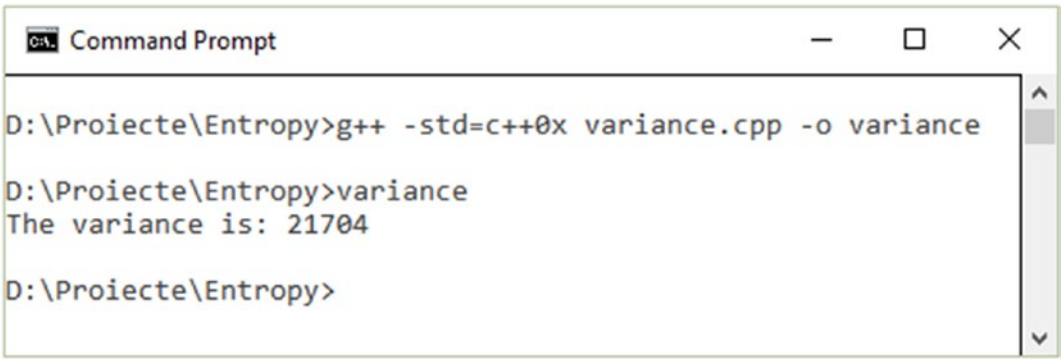


Figure 3-4. *Output of the variance*

Listing 3-4. Source Code

```
#include<iostream>

using namespace std;

//** the below function is used
//** for computing the variance
int computingVariance(int n[], int h)    //**a=n, n=h
{
    //** will compute the mean
    //** average of the elements
    int sum = 0;
    for (int k = 0; k < h; k++)
        sum += n[k];
    double theMean = (double)sum /
                     (double)h;

    //** calculate the sum squared
    //** differences with the mean
    double squared_differences = 0;
    for (int t=0; t<h; t++)
        squared_differences += (n[t] - theMean) *
                               (n[t] - theMean);
    return squared_differences / h;
}

int main()
{
    int arr[] = {600, 470, 170, 430, 300};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The variance is: "
         << computingVariance(arr, n) << "\n";
    return 0;
}
```

Case Study 5: Computing the Standard Deviation

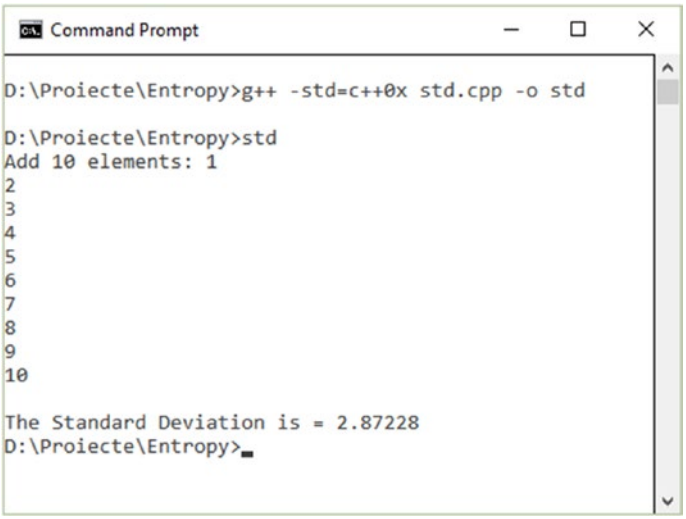


Figure 3-5. *Output of the standard deviation*

Listing 3-5. Source Code

```
#include <iostream>
#include <cmath>

using namespace std;

float computeStandardDeviation(float data[]);

int main()
{
    int n;
    float elements_array[10];
```

```
    cout << "Add 10 elements: ";
    for(n = 0; n < 10; ++n)
        cin >> elements_array[n];

    cout << endl << "The Standard Deviation is = " <<
                    computeStandardDeviation(elements_array)<<endl;

    return 0;
}

float computeStandardDeviation(float elements_array[])
{
    float theSum = 0.0, theMean, theStandardDeviation = 0.0;

    int j,k;

    for(j = 0; j < 10; ++j)
    {
        theSum += elements_array[j];
    }

    theMean = theSum/10;


    for(k = 0; k < 10; ++k)
        theStandardDeviation += pow(elements_array[k] -
                                    theMean, 2);

    return sqrt(theStandardDeviation / 10);
}
```

Case Study 6: Birthday Paradox

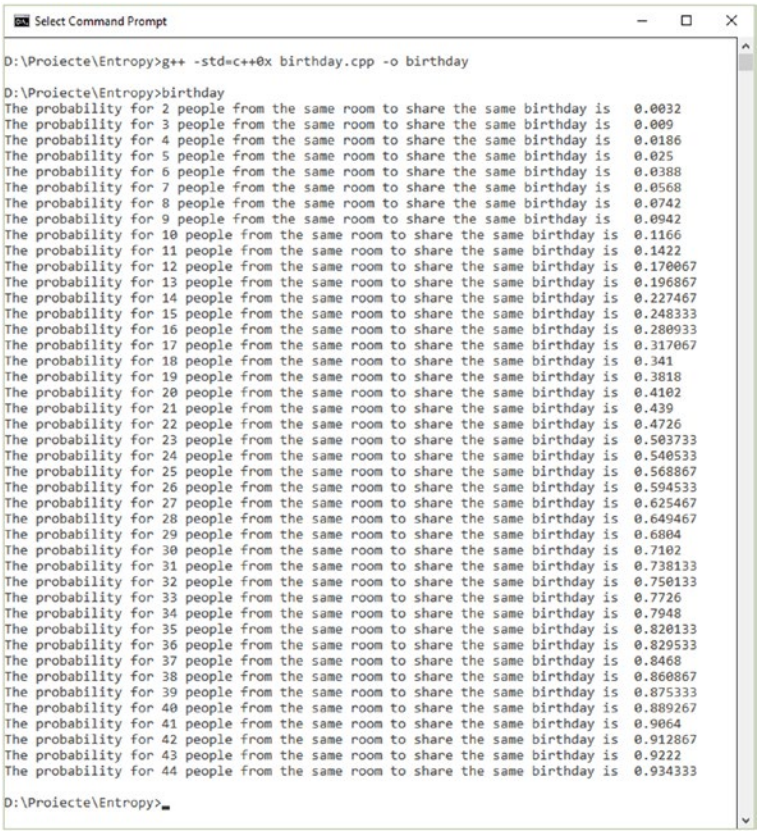


Figure 3-6. *Output of the birthday computation*

Listing 3-6. Source Code

```
#include <ctime>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, const char *argv[])
{
   const int processes = 15000;
   short int no_of_birthdays[365];
   int processesWithSuccess;
```

```
   bool IsSharedBirthday;

   //** we will time(NULL) as seed to be used for the
   //** pseudo-random number generator srand()
   srand(time(NULL));

   for (int no_of_people=2;no_of_people<45;
++no_of_people)
   {
      processesWithSuccess = 0;
      for (int i = 0; i < processes; ++i)
      {
          //** all birthdays will be set to 0
         for (int j=0;j<365;no_of_birthdays[j++] = 0);
         IsSharedBirthday = false;
         for (int j = 0; j < no_of_people; ++j)
         {
            //** if our given birthday is shared (this
 //** means that is assigned for more than one
 //** person) this will be a shared birthday
 //** and we will need to stop verifying.
            if (++no_of_birthdays[rand() % 365] > 1){
               IsSharedBirthday = true;
               break;
            }
         }
         if (IsSharedBirthday) ++processesWithSuccess;
      }

      cout << "The probability for " << no_of_people << "people from the
      same room to share the same birthday is \t"<<(float(processesWithSucc
      ess)/ float(processes))<<endl;
   }
   return 0;
}
```

Case Study 7: (Extended) Euclidean Algorithm

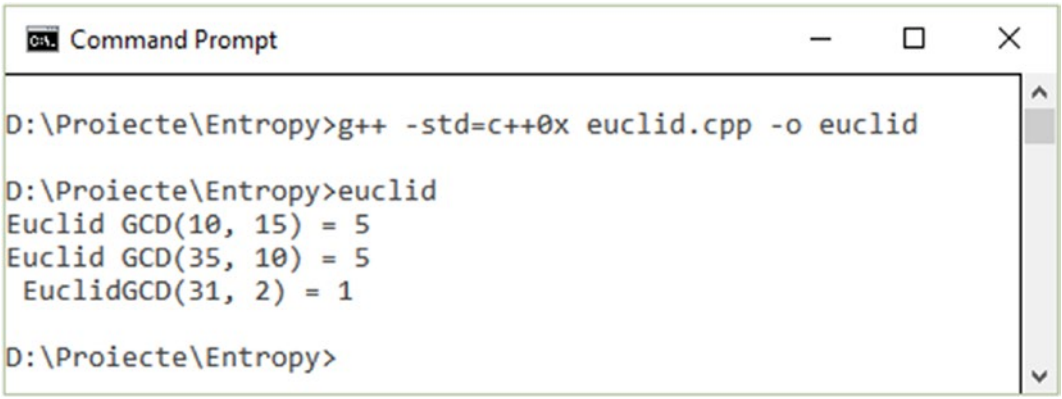


Figure 3-7. *Output of the Euclidean Algorithm*

Listing 3-7. Source Code

```
//** NOTE: bits/stdc++ does not represent
//** a standard header file of GNU C++ library.
//** If the code will be compiled with other
//** compilers than GCC it will fail
#include<stdio.h>

using namespace std;

//** the function will compute
//** the GCD for two number
int g(int x, int y)  {
    if (x == 0)
        return y;
    return g(y % x, x);
}

int main()
{
    int x = 10, y = 15;
    cout << "Euclid GCD(" << x << ", "
         << y << ") = " << g(x, y)
                        << endl;
```

```
    x = 35, y = 10;
    cout << "Euclid GCD(" << x << ", "
         << y << ") = " << g(x, y)
                        << endl;
    x = 31, y = 2;
    cout << " EuclidGCD(" << x << ", "
         << y << ") = " << g(x, y)
                        << endl;
    return 0;
}
```

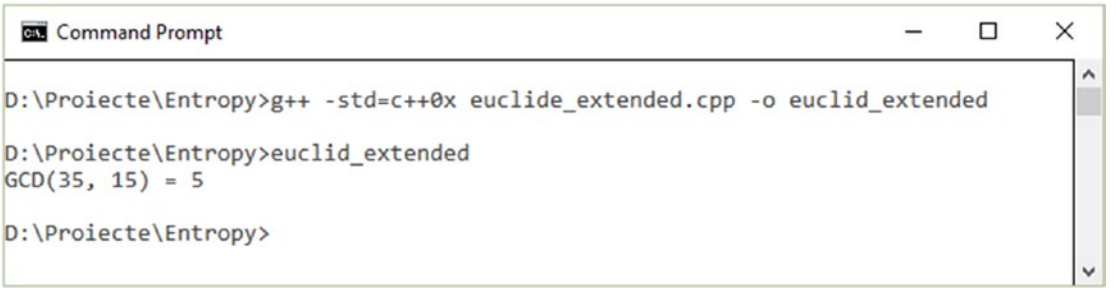


Figure 3-8. *Output of the extended Euclidean algorithm*

Listing 3-8. Source Code

```
#include <bits/stdc++.h>
using namespace std;

//** computing extended euclidean algorithm
int g_e(int x, int y, int *w, int *z)
{
    //** this is the basic or ideal case
    if (x == 0)
    {
        *w = 0;
        *z = 1;
        return y;
    }
```

```
     //** variables for storing the results
     //** for the recursive call
    int a1, b1;
    int g = g_e(y%x, x, &a1, &b1);

    //** with help of the recursive call
      //** update a and b with the results
    *w = b1 - (y/x) * a1;
    *z = a1;

    return g;
}

// Driver Code
int main()
{
    int a, b, w = 35, y = 15;
    int g = g_e(w, y, &a, &b);
    cout << "g_e(" << w << ", " << y<< ") = " << g << endl;
    return 0;
}
```

Case Study 8: Computing the Multiplicative Inverse Under Modulo *q*

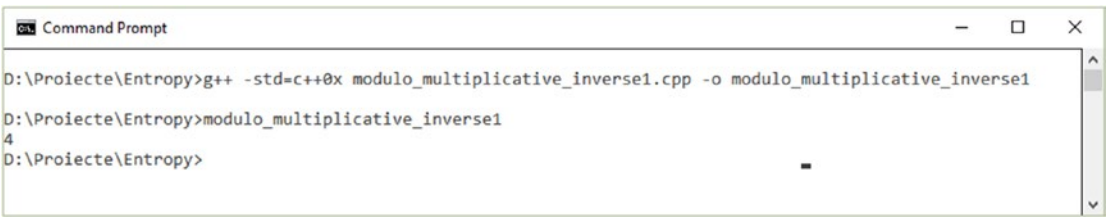


Figure 3-9. *Output of the modular multiplicative inverse (a basic and tricky form of the implementation)*

Listing 3-9. Code for Computing the Modular Multiplicative Inverse (Tricky Method)

```
#include<iostream>
using namespace std;

//** this represents the basic method or tricky method
//** for finding modulo multiplicative inverse of
//** x under modulo m
int modulo_inverse(int x, int m)
{
    x = x%m;
    for (int y=1; y<m; y++)
       if ((x*y) % m == 1)
          return y;
}

int main()
{
    int x = 3, m = 11;
    cout << modulo_inverse(x, m);
    return 0;
}
```

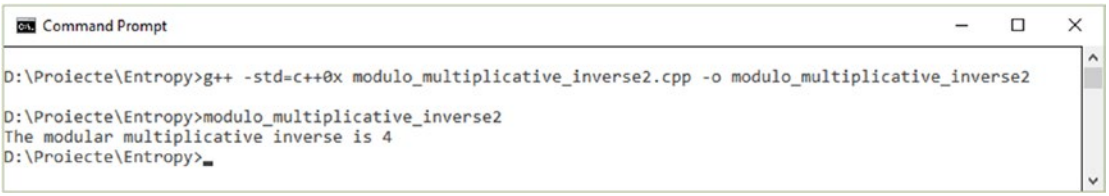


Figure 3-10. *Output of the modular multiplicative inverse (when the number is coprime)*

Listing 3-10. Source Code

```
#include<iostream>
using namespace std;

//** function for computing extended euclidean algorithm
int gcd_e(int x, int y, int *w, int *z);
```

```
void modulo_inverse(int h, int modulo)
{
    int i, j;
    int g = gcd_e(h, modulo, &i, &j);
    if (g != 1)
        cout << "There is no inverse.";
    else
    {
         //** we add the modulo in
        //** order to handle negative i
        int result = (i%modulo + modulo) % modulo;
        cout << "The modular multiplicative inverse is " <<
                                                    result;
    }
}

//** we will compute the extended euclidean algorithm
int gcd_e(int h, int k, int *w, int *z) {
    //** the "happy" case
    if (h == 0){
        *w = 0, *z = 1;
        return k; }

     //** storing results of our recurive invoke
    int a1, b1;    //** x1=a1, y1=b1
    int g = gcd_e(k%h, h, &a1, &b1);

    //** with recursive invocation results
    //** we will update x and y
    *w = b1 - (k/h) * a1;
    *z = a1;

    return g;
}
```

```
int main()
{
    int x = 3, modulo = 11;
    modulo_inverse(x, modulo);
    return 0;
}
```

Case Study 9: Chinese Remainder Theorem

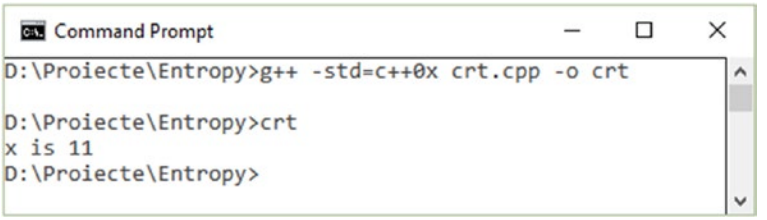


Figure 3-11. *Ouput for the Chinese Remainder Theorem*

Listing 3-11. Source Code

```
#include<iostream>

using namespace std;

int inverse(int x, int modulo)
{
    int modulo0 = modulo, k, quotient;
    int a0 = 0, a1 = 1;

    if (modulo == 1)
       return 0;

    //** we will apply extended euclidean algorithm
    while (x > 1)
    {
        quotient = x / modulo;

        k = modulo;

        //** modulo represents the remainder
        //** continue with the process same as
```

```
        //** euclid's algorithm
        modulo = x%modulo, x=k;

        k = a0;

        a0 = a1 - quotient * a0;

        a1 = k;
    }

    //** make a1 positive
    if (a1 < 0)
       a1 += modulo0;

    return a1;
}

int lookForMinX(int numbers[], int remainders[], int l)
{
    //** computing the product for all the numbers
    int product = 1;
    for (int j = 0; j < l; j++)
        product *= numbers[j];

    //** we initialize the result with 0
    int result = 0;

    //** apply the formula mentioned above
    for (int j = 0; j < l; j++)
    {
        int pp = product / numbers[j];
       result += remainders[j] * inverse(pp, numbers[j]) * pp;
    }

    return result % product;
}

int main(void) {
    int numbers[] = {3, 4, 5};
    int remainders[] = {2, 3, 1};
    int k = sizeof(numbers)/sizeof(numbers[0]);
```

```
    cout << "x is " << lookForMinX(numbers, remainders, k);
    return 0;
}
```

Case Study 10: The Legendre Symbol

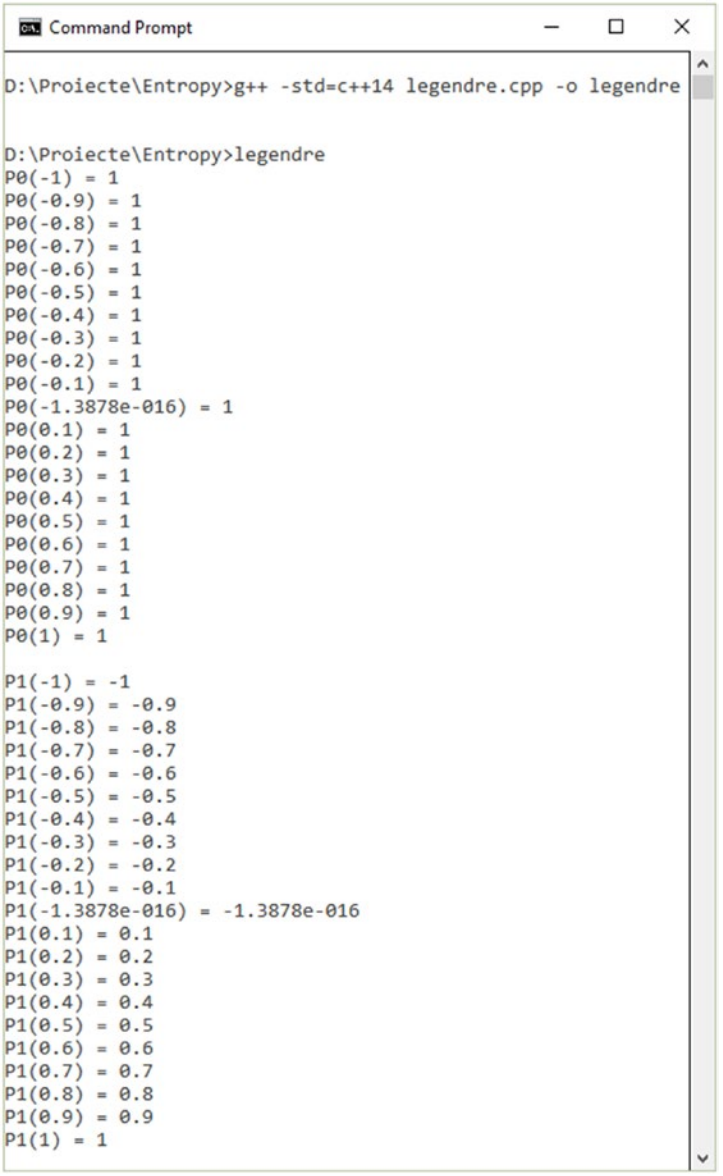


Figure 3-12. *Output of the Legendre symbol*

The source code for the implementation of the Legendre symbol is structured in two files:

- `legendre.cpp` (see Listing 3-12)
- `legendre.h` (see Listing 3-13)

To compile the source code, the following command needs to be run:

```
g++ -std=c++2a legendre.cpp –o legendre
```

Listing 3-12. Source Code (legendre.cpp)

```
#include <iostream>
#include "legendre.h"

using namespace std ;
using namespace LegendreStorage::Legendre ;

int main()
{
  double p_n;

  cout.precision(5) ;
  for (unsigned int v = 0 ; v <= 5 ; v++)
  {
    for (double b = -1.0 ; b <= 1.0 ; b = b + 0.1)
    {
      p_n = Polynom_n<double>(v, b) ;
      cout << "P" << v << "(" << b << ") = " << p_n << endl ;
    }
    cout << endl ;
  }

  return 0 ;
}
```

Listing 3-13. Source Code for the Legendre Symbol (legendre.h)

```
#ifndef __LEGENDRESYMBOL_H__
#define __LEGENDRESYMBOL_H__

namespace LegendreStorage {
  namespace Legendre{
    //** when n=0
    template <class T> inline auto Polynom0(const T& x){
      return static_cast<T>(1);
    }

    //** when n=1
    template <class T> inline auto Polynom1(const T& x){
      return x;
    }

    //** when n=2
    template <class T> inline auto Polynom2(const T& x){
      return ((static_cast<T>(3)*x*x) -
       static_cast<T>(1)) / static_cast<T>(2);
    }

    //** polynom(x)
    template <class T> inline auto Polynom_n(unsigned int h,
                                             const T& y)
    {
      switch(h){
        case 0:
          return Polynom0<T>(y);

        case 1:
          return Polynom1<T>(y);

        case 2:
          return Polynom2<T>(y);

        default:
          break;}
```

```
      auto polynom_1(Polynom2<T>(y));
      auto polynom_2(Polynom1<T>(y));
      T polynom;

      for (auto a=3u; a<=h; ++a){
         polynom = ((static_cast<T>((2 * a) - 1)) * y *
                                          polynom_1
         - (static_cast<T>(a - 1) * polynom_2)) /
                                  static_cast<T>(a);
        polynom_2 = polynom_1;
        polynom_1 = polynom;      }

      return polynom;   }}}
#endif
```

Conclusion

This chapter discussed the importance of some mathematical tools that are used in most modern cryptography algorithms. We showed how they can be implemented and we explained the important steps of the algorithms. We also covered the important aspects of mathematical foundations, such as the probability theory, information theory, number theory, and finite fields.

For each mathematical foundation, we presented the necessary equations and mathematical expressions that are used in the implementation of the algorithms. Each equation or mathematical expression was demonstrated through a software application implemented in C++, entitled as a case study. Each case study demonstrated the skills and knowledge required by you in order to develop secure and reliable code. By reaching to the end of the chapter, you should now understand of the important notions and terms, the programming concepts and algorithms used, both theoretical and practical, and how to move from theory to practice in a very short time.

References

[1] Alfred J. Menezes, Paul van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*. CRC Press. ISBN 0-8493-8523-7. 1996.

[2] A. R. Meijer, *Algebra for Cryptologists*, first edition, 2016 edition. New York, NY: Springer, 2016.

[3] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*, second edition, 2014 edition. New York: Springer, 2014.

[4] S. Rubinstein-Salzedo, *Cryptography*, 1st ed. 2018 edition. New York, NY: Springer, 2018.

[5] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 6th edition. USA: Prentice Hall Press, 2013.

[6] Khan Academy, *Cryptography: Data and Application Security*. Independently published, 2017.

[7] C. T. Rivers, *Cryptography: Decoding Cryptography! From Ancient To New Age Times*. JR Kindle Publishing, 2014.

[8] D. Stinson, *Cryptography: Theory and Practice*, Second edition. CRC/C&H, 2002.

[9] H. Delfs and H. Knebl, *Introduction to Cryptography: Principles and Applications*, third edition, 2015 edition. New York, NY: Springer, 2015.

[10] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, second edition. Boca Raton: Chapman and Hall/CRC, 2014.

[11] X. Wang, G. Xu, M. Wang, and X. Meng, *Mathematical Foundations of Public Key Cryptography*, first edition. Boca Raton: CRC Press, 2015.

[12] T. R. Shemanske, *Modern Cryptography and Elliptic Curves. Providence*, Rhode Island: American Mathematical Society, 2017.

[13] S. Y. Yan, *Primality Testing and Integer Factorization in Public-Key Cryptography*, first edition. Springer, 2013.

[14] L. M. Batten, *Public Key Cryptography: Applications and Attacks*. Hoboken, N.J: Wiley-Blackwell, 2013.

[15] J.-P. Aumasson, *Serious Cryptography*. San Francisco: No Starch Press, 2017.

[16] S. Khare, *The world of Cryptography: incl. cryptosystems, ciphers, public key encryption, data integration, message authentication, digital signatures.*

[17] Adrian Atanasiu, *Securitatea Informației (Information Security) - Criptografie (Cryptography) - volume 1*, InfoData Publisher, ISBN: 978-973-1803-29-6, 978-973-1803-16-6. 2007. [Romanian Language]

[18] Adrian Atanasiu, *Securitatea Informației (Information Security) – Protocoale de Securitate (Security Protocols) - volume 2*, InfoData Publisher, ISBN: 978-973-1803-29-6, 978-973-1803-18-0. 2007. [Romanian Language]

[19] *Vasile Preda, Emil Simion, and Adrian Popescu, Criptanaliza. Rezultate şi Tehnici Matematice (Cryptanalysis. Results and Mathematical Techniques), Universitatea Bucuresti Publisher, ISBN: 973-575-975-6. 2004.* [Romanian Language]

CHAPTER 4

Large Integer Arithmetic

This chapter will cover the arithmetic operations and explain how to work with large integers. Some cryptographic algorithms require big integers that don't fit within the normal size of variables, such as `int`. We will give a quick overview of big integers and some of the libraries that are used to work with them.

In implementing complex cryptography algorithms, the operations with large integers can be very difficult to perform. The limitations can be due to hardware equipment (e.g. processor, RAM memory) or programming languages.

In C/C++, an integer is represented as 32 bits. Out of the 32 bits, only 31 can be used to represent positive integer arithmetic. In cryptography, we can deal with numbers that are up to two billion, $2 \cdot 10^9$.

Some compilers, such as GNU C++ or g++, offer a long long type. This provides the ability to represent integers around 9 quintillion, $9 \cdot 10^{18}$. For most simple cryptographic operations, this is good, but some cryptographic algorithms require more digits in their integer representation. Let's consider as an example the RSA (Rivest-Shamir-Adleman) public-key encryption cryptosystem, which requires around 300 digits. If we are dealing with specific real events and their probabilities, the computation often will involve numbers that are very large. The output and achieving the main result might be appropriate for C/C++. Compared with other complex computations, we will have very large numbers.

As an interesting example, consider the chances of winning the lottery jackpot with one ticket. The combinations are of 50 taken 6 at a time, "50 choose 6" is $\frac{50!}{((50-6)!\cdot 6!)}$.

The resulting number is 15.890.700, so the chances for winning are 1/15.890.700. Using the C/C++ programming language, the number 15.890.700 can be easily represented. However, this could be tricky and we could easily fall for naiveté during the implementation of 50! (computed using Calculator from Windows), which is 3.041409320171e+64 or 30,414,093,201,713,378,043,612,608,166,064,768,844,377,641,568,960,512,000,000,000,000

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_4

Using C/C++ to represent that number will be almost impossible, even on a 64-bit platform.

Big Integers

In the following sections, we will examine a couple of algorithms that can be used for arithmetic operations using *big* integers. Remember, when working with cryptography algorithms and security mechanisms, implementation can be very tricky when dealing with big integers. Below, we will show a step-by-step methodology of how to work with big numbers.

We will transform a standard integer using different computations in a big integer. In order to accomplish this, we will write a function named `transformIntToBigInt(A, 123)`. The function goal is to initialize A as `A[0]=3`, `A[1]=2`, `A[2]=1`, and zeroes for the remaining positions as `A[3,...N-1]`. Listing 4-1 shows to accomplish the statement from above by using a simple implementation in C/C++. The *BASE* represents the bit sign.

Listing 4-1. Transforming a Standard Integer Using Different Computations in a Big Integer[1]

```
void transformIntToBigInt(int BigNo[], int number)
{
     Int k;
     int bitSign;
     int BASE;

     //** start indexing with 0 position
     k = 0;

     //** if we still have something left
     //** within the number, continue
     while (number) {
          //** insert the digit that is least significant
          //** into BigNo[k] number
```

[1]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

```
            BigNo[k++] = number % bitSign;

            //** we don't need the least significant bit
            number /= BASE;
        }

        //** complete the remain of the array with zeroes
        while (k < N)
           BigNo[k++] = 0;
}
```

The algorithm from Listing 4-1 has O(N) space and time.

Let's continue our journey by looking at the possibility of adding 1 to a big int. This is a very useful operation and it is quite frequently used in cryptography. The advantage is that it is much easier than the full addition. See Listing 4-2.

Listing 4-2. Adding 1 to a big int[2]

```
void increment (int BigNo [])
{
    Int  i;
    int N;
    int BASE;

    //* start indexing with least significant digit

    i = 0;
    while (i < N)
    {

        //* increment the digit
        BigNo[i]++;

        //** if it overflows
        if (BigNo[i] == BASE)
```

[2]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

```
        {
              //** make it zero and move the index to next
              BigNo[i] = 0;
              i++;
        }
        else
           //** else, we are done!
           break;
    }
}
```

The algorithm illustrated in Listing 4-2 takes $O(n)$ for the worst case possible (just imagine something like 999999999999999999999999....) and $\Omega(1)$ for the best case. The best case is when we don't have any overflow on the least significant digit.

Moving forward, let's look at a method for adding two big integers. In this case, we want to add two big integers in two different arrays, `BigNo1[0,..., N-1]` and `BigNo2[0,...,N-1]`. The output result will be saved in another array, `BigNo3[0,...,N-1]`. The algorithm is quite basic; there is nothing fancy about it. See Listing 4-3.

Listing 4-3. Addition Algorithm[3]

```
void addition(int BigNo1[], int BigNo2[], int BigNo3[])
{
    Int  j, overflowCarry, sum;
    int carry, N, BASE;

    //** There is no need to carry yet
    carry = 0;

    //** move from the least to the most significant digit
    for (j=0; j<N; j++)
```

[3]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

```
    {
        //** the digit from j'th position of BigNo3[]
        //** represents the sum of j'th digits of
        //** BigNo1[] and BigNo2[] plus the overvflow carry
        sum = BigNo1[j] + BigNo2[j] + overflowCarry;

        //** if the sum will go out of the base then
        //** we will find ourself in an overflow situation
        if (sum >= BASE)
        {
               carry = 1;

               //** adjust in such way that
               //** the sum will fit within a digit
               sum -= BASE;
        }
        else
              //** otherwise no carryOverflow
              carry = 0;

        //** add the result in the same sum variable
        BigNo3[j] = sum;
     }

     //** if we are getting to the
     //** end we can expect an overflow
     if (carry)
         printf ("There is an overflow in the addition!\n");
}
```

Let's continue with multiplication. We will use a basic method to multiply two large numbers, `X` and `Y`, multiplying each digit of `X` with each digit of `Y`, so the output will be a partial product. The output result will be shifted to the left for every new digit. Function `multiplyingOneDigit` will multiply an entire big integer with a single digit. The result will be placed in a new large integer. Function `left_shifting` will shift the number to the left a certain number of spaces. It will be multiplied using b^i, where `b` is base and `i` represents the numbers of spaces. Let's have a quick look at the algorithm, which is shown in Listing 4-4.

Listing 4-4. Multiplication[4]

```
void multiply (int BigInt1[], int BigInt2[], int BigInt3[])
{
     int length_of_integer;
     int  x, y, P[length_of_integer];

     //** C will store the sum of
     //** partial products.  It's initially 0.
     transformIntToBigInt (BigInt3, 0);

     //* for each digit in BigInt1
     for (x=0; x<length_of_integer; x++)
     {
          //** multiply BigInt2 by digit [x]
          multiplyUsingOneDigit (BigInt2, P, BigInt1[x]);

          //** left shifting the partial product with i bytes
          leftShifting(P, x);

          //** add the output result to the current sum
          addResult(BigInt3, P, BigInt3);
     }
}
```

Moving forward, we will examine a function whose purpose is to multiply by a single digit. See Listing 4-5.

[4]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

Listing 4-5. Multiplying Using a Single Digit[5]

```
void multiplyUsingOneDigit (int BigOne1[], int BigOne2[],
                                                    int number) {
     int  k, carryOverflow;
     int N, BASE;

      //** there is nothing related to
      //** extra overflow to be added at this moment
      carryOverflow = 0;

     //** for each digit, starting with least significant...

     for (k=0; k<N; k++){
          //** multiply the digit by number,
          //** putting the result in BigOne2
          BigOne2[k] = number * BigOne1[k];

          //** adding extra any overflow that is taking
          //** place starting with the last digit
          BigOne2[k] += carryOverflow;

          //** product is too big to fit in a digit
          if (BigOne2[k] >= BASE) {
               //** handle the overflow
               carryOverflow = BigOne2[k] / BASE;
               BigOne2[k] %= BASE;
          }
          else
               //** no overflow
               carryOverflow = 0;
     }
     if (carryOverflow)
          printf ("During the multiplication
                  we experienced an overflow!\n");
}
```

[5]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

We will continue with a functional that will shift to the left a specific number of spaces, as shown in Listing 4-6.

Listing 4-6. Shifting to the Left a Specific Number of Spaces[6]

```
void leftShifting (int BigInt1[], int number) {
     int i;

     //** moving starting from left to right,
     //** we will move anything with left n spaces
     for (i=N-1; i>= number; i--)
          BigInt1[i] = BigInt1[i- number];

     //** complete the last n digits with zeros
     while (i >= 0) BigInt1[i--] = 0;
}
```

Big Integer Libraries

There are several libraries and frameworks that deal with high numbers. For some, the development process was suspended, but they are still used in cryptography applications.

The libraries for working with big integers are as follows:

- **Matt McCutchen**[7] proposed a very easy-to-use C++ library for calculations on big integers [1]. The code has very good explanations and it is easy to follow. The results obtained in symmetric and asymmetric cryptography algorithms were promising. Most of the results were compared with other tools for reference and checking, such as CryptTool.[8]

[6]The code is meant to be a sketch of a function that will transform a simple integer to a big integer. The source code will not compile without a proper adjustment for a real cryptographic application.

[7]Matt McCutchen's web site, https://mattmccutchen.net/

[8]CrypTool, www.cryptool.org/en/

- **L3HARRIS Geospatial Solutions** offers the Big Integer Class [2], which is another library that is fast on computations.[9]
- **Boost Library**[10] is another strong library used to achieve tasks based on linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. The library has an impressive set of independent libraries (around 160) and the documentation is well structured and quite easy to follow and use [3].
- **GMP Library** (GNU Multiple Precision Arithmetic Library) is another free library that can be used for random precision arithmetic.[11] It offers support for operations based on signed integers, rational numbers, and floating point numbers (see Chapter 6 for more details). The only limitation of the library involves the available memory. The limits are $2^{32} - 1$ bits on 32-bit systems and $2^{37} - 1$ bits on 64-bit systems. The main interface is for C/C++, but there is also support for C#, .NET, and OCaml. (It can easily be ported for Haskell as well. For more details, take a look at [4], [5], and [11]). Also, there is important support for Ruby, PHP, Python, R, Perl, and the Wolfram Language. The main goals and targets of the library are cryptography software applications, security of the Internet, and algebra systems.
- **LibBF Library [8]** is used for working with floating point numbers represented in base 2. The library is based and implemented on the IEEE 754 standard [7]. The example provided on the library web page, TinyPI, is a very good example to show its power. This library will be examined further in Chapter 6.
- **Bignum C++ Library [9],** TTMath, allows both personal and commercial users to perform arithmetic operations. The types of integers supported are big unsigned integers, big signed integers, and biplying g floating point numbers. There is support for mathematical operations, such as adding, subtracting, dividing, and multiplying. See Listing 4-7.

[9]L3HARRIS Geospatial Solutions, www.harrisgeospatial.com

[10]Boost Library, www.boost.org

[11]GMP Library, http://gmplib.org

Listing 4-7. Using ttmath::UInt<>

The current example, which is described as well on [10], it will create an object characterized by two words each. On a 32-bit platform the maximum value that can be held is $2^{32*2} - 1$. Take note of the fact that the author shows that variables can be intitialized with string or if we are dealing with small values is using a standard type such as unsigned int.

```
#include <ttmath/ttmath.h>
#include <iostream>

int main()
{
     ttmath::UInt<2> firstA, secondB, thirdC;

     a = "8765";
     b = 3456;
     c = a*b;

     std::cout << thirdC << std::endl;
}<>
```

Conclusion

The chapter discussed the general representation of big integers and their operations. We analyzed the most important big integer libraries, highlighting the advantages of the libraries for you when you are setting up an environment for developing cryptographic algorithms.

References

[1] C++ Big Integer Library. Available online: https://mattmccutchen.net/bigint/. Last access: May 16, 2020.

[2] BigInteger Class L3HARRIS Geospatial. Available online: www.harrisgeospatial.com/docs/BIGINTEGER.html.

[3] Boost Library Documentation. Available online: www.boost.org/doc/libs/1_72_0/.

[4] S.L. Nita and M. Mihailescu, *Practical Concurrent Haskell: With Big Data Applications*. Apress. 2017.

[5] S.L. Nita and M. Mihailescu, *Haskell Quick Syntax Reference*. Apress. 2019.

[6] Bellard. Available online: https://bellard.org/libbf/.

[7] IEEE 754-2019 – Standard for Floating-Point Arithmetic. Available online: https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html.

[8] LibBF Library. Available online: https://bellard.org/libbf/.

[9] Bignum Library. Available online: www.ttmath.org/.

[10] TTMath Samples. Available online: www.ttmath.org/samples.

[11] A.S. Mena, *Practical Haskell: A Real World Guide to Programming*. Apress. 2019.

CHAPTER 5

Floating-Point Arithmetic

As discussed in the previous chapter, working with big integers is an abstract art, and if the schemes are not implemented properly, the entire cryptographic algorithm or scheme can lead to a real disaster.

This chapter is dedicated to floating-point arithmetic and its importance for cryptography.

Why Floating-Point Arithmetic?

Floating-point arithmetic represents a special type of arithmetic which requires caution due to the representations and methods of implementations. This type of arithmetic can be observed in chaos-based cryptography or homomorphic encryption, which is presented later in *Chapter 14 and Chapter 12*, respectively.

Computations using floating-point numbers can be found within systems that use small and very large real numbers. The process must be very fast during the computations.

A floating point variable is a special type of variable that is able to hold real numbers, such as 5420.0, − 4.213,or 0.045634. The *floating* part means that the decimal point can "float."

C++ offers different floating point data types, such as *float, double,* and *long double.* As you know from C++ and integers, the language does not define any size for these types. With modern architectures, most of the floating point representations are with respect for the IEEE 754 standard for the binary representation format. According to this standard, a float type has 4 bytes, a double has 8 bytes, and a long double has 8 bytes (same as the double), and it represents the 80-bit extended precision for the x86 architectures (by padding, we have 12 bytes or 16 bytes).

When you work with floating values, always make sure that you include at least one decimal. This will help the compiler understand the difference between a floating-point number and an integer. Actually, this is very important for cryptographers.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_5

```
int a{4};           //** 4 is an integer
double b{3.0};      //** 3.0 represents a floating point (with no
                    //** suffix – double type by default)
float c{6.0f};      //** 6.0 represents a floating point (f is the
                    //** suffix which means a float type)
```

Displaying Floating Point Numbers

Let's consider the example in Listing 5-1.

Listing 5-1. Displaying Common Float Numbers

```
#include <iostream>

using namespace std;

int main()
{
        cout << 5.0 << endl;
        cout << 6.7f << endl;
        cout << 9876543.21 << endl;

        return 0;
}
```

The output is shown in Figure 5-1.

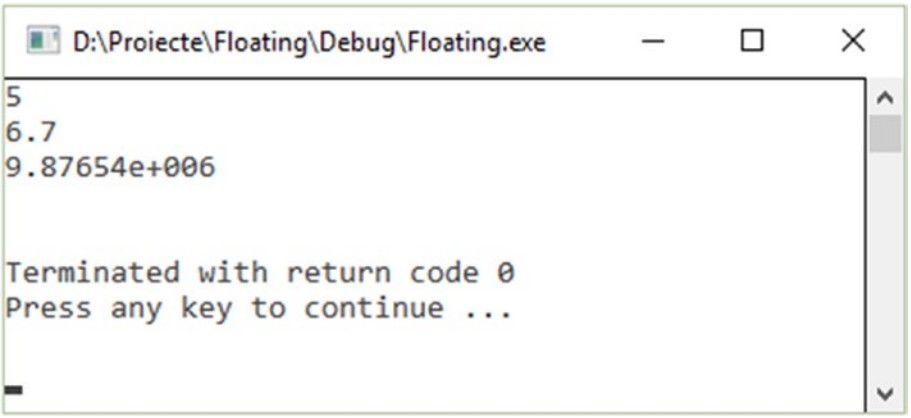


Figure 5-1. *The output of common float numbers*

By looking on the output of the program, you can observe that in the first case, the output is 5 but the source code shows 5.0. This is happening because the fractional part is equal to 0. In the second case, the number printed is identical to the one from the source code. In the third case, the number is displayed using scientific notation, which is an important asset for cryptography algorithms.

The Range of Floating Point Numbers

Let's have a look at the IEEE 754 representation and consider the following sizes with their range and precision. See Table 5-1.

Table 5-1. *IEEE 754 Standard Representation*

Size	Range	Precision
4 bytes	$\pm 1.18 \times 10^{-38}$ *to* $\pm\ 3.4 \times 10^{38}$	6-9 are the most important digits. Usually around 7 digits.
8 bytes	$\pm 2.23 \times 10^{-308}$ *to* $\pm\ 1.80 \times 10^{308}$	15-18 are most important digits. Usually around 16 digits.
80-bits (usually using 12 or 16 bytes)	$\pm 3.36 \times 10^{-4932}$ *to* $\pm\ 1.18 \times 10^{4932}$	18-21 are most important digits.
16 bytes	$\pm 3.36 \times 10^{-4932}$ *to* $\pm\ 1.18 \times 10^{4932}$	33-36 are most important digits.

The 80-bit floating point on today's processors is implemented using 12 or 16 bytes. The processors can handle this size easily.

Floating Point Precision

Let's consider the following example represented by the fraction $\frac{1}{3}$. The decimal representation is 0.3333333… with an infinity of 3*s*.

Using a computer, a number with an infinite length would require infinite memory to store it. This limitation of the memory restricts the storage of a floating point number to a specific number of important digits. A floating point number and its precision define how many important digits can be represented without any information loss.

In cryptography, if we are outputting a floating point number, `cout` has an implicit precision of 6. In Figure 5-2, you can see how `cout` in Listing 5-2 truncates the values to six digits.

Listing 5-2. Representation of Floating Point Precision

```
#include <iostream>

using namespace std;

int main()
{
    cout << 7.56756767f << endl;
    cout << 765.657667f << endl;
    cout << 345543.564f << endl;
    cout << 9976544.43f << endl;
    cout << 0.00043534345f << endl;

    return 0;
}
```

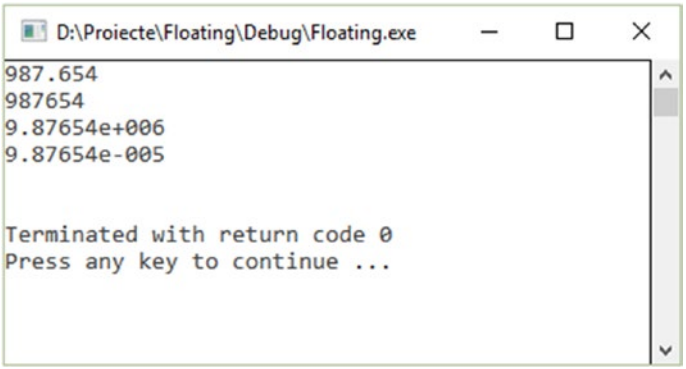


Figure 5-2. *Output of floating point precision*

Remember that each of the cases from above will have only six important digits.

Note the fact that with `std::cout` in some of the cases the output will be represented using scientific notation. According to the compiler used, the exponent will usually be padded within a minimum number of digits. The number of digits for the exponent

that are displayed is based on the compiler used. For example, Visual Studio uses three and other compilers use two (they are implemented according to C99 instructions and standards).

The number of digits that represents floating point number and its precision are dependent on both the size and the specific value that is stored. The float values are represented with 6 and 9 digits as precision, with most values having a minimum of 7 important digits. The double values are represented with 15 and 18 digits as precision. Long double values are represented with at least a precision of 15 or 33 important digits, which are dependent on how the bytes are occupied.

The code in Listing 5-3 overrides the default precision that `cout` or `std::cout` displays by using the `setprecision()` function. The `setprecision()` function is defined within the `iomanip` header. See Figure 5-3 for the output.

Listing 5-3. Default Precision

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
     std::cout << std::setprecision(16);
     std::cout << 3.33333333333333333333333333333333f <<endl;
     std::cout << 3.33333333333333333333333333333333 << endl;

     return 0;
}
```

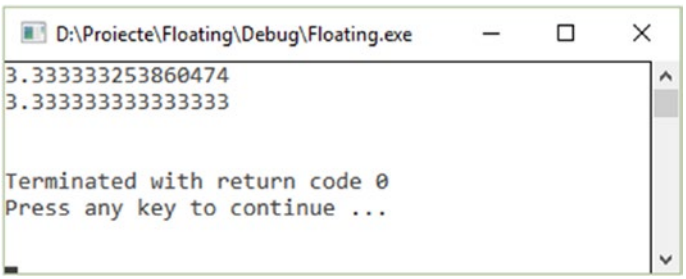


Figure 5-3. *Overriding the default precision*

In the above example, we set the precision to 16 digits. Each of the numbers is shown with a precision of 16 digits. The issues raised by the precision will not just impact the fractional number; they will impact any number that has multiple important digits.

Next Level for Floating-Point Arithmetic

In Chapter 12, we will introduce a complex type of encryption. Homomorphic encryption is a special type of encryption that is used as a professional technology for privacy preserving and it outsources storage and computation. This type of encryption allows data to be encrypted and outsourced to commercial (or public) environments for processing purposes, all while the data are encrypted. Homomorphic encryption is derived from ring learning with errors (see Chapter 13) and related to private set intersection [1].

Moving on to complex cryptosystems, floating-point representation represents the core of the encryption/decryption mechanisms, finding the proper way to approximate a real number in such a way as to support a compromise between range and precision.

As mentioned, the "floating" term means that a number's decimal point can float. This means that it can be set anywhere related to the important digits of the number. To be more exactly, when dealing with complex cryptosystems, such as homomorphic encryption, floating-point number a can be shown as four integers, such as

$$a = \pm d \cdot n^{f-j}$$

where n represents the base, f represents the exponent, j represents the precision, and d represents the important or significand that must satisfy the following relation:

$$0 \le d \le n^f - 1$$

C++ offers floating-point manipulation with the functions `fmod`, `remainder`, and `remquo`. These functions can be found within the `cmath` header file. Starting with C++11, these basic functions are used for handling simple mathematical computations related to floating point numbers that are necessary for common programming and for cryptography (low and simple concepts). For advanced cryptography algorithms, the functions are quite limited and they don't provide the necessary equipment for a cryptographer. To achieve complex computations with big real numbers, you can use

professional libraries such as Boost Multiprecision Library, TTMath, LibBF, GNU Multi-Precision Library, and more. They will help you achieve complex tasks with complex cryptosystems.

Conclusions

This chapter discussed the general representations of floating point numbers and how they are used in complex cryptosystems. We explained the importance of floating-point arithmetic for complex cryptosystems, such as homomorphic encryption, chaos-based cryptography, lattice-based cryptography or ring learning with errors. Without a proper understanding of floating-point arithmetic, advanced cryptosystems cannot be implemented properly. A wrong implementation can lead to a huge disaster for a commercial cloud computing or a big data environment.

References

[1] H. Chen, K. Laine, and P. Rindal, *Fast Private Set Intersection from Homomorphic Encryption*. 2018.

[2] Leo Ducas and Daniele Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–Eurocrypt 2015*. pp. 617–640. Springer; Berlin, Germany. 2015.

[3] S. Halevi and V. Shoup, "Algorithms in HElib," in *Crypto'14*, vol. 8616. Springer; Heidelberg, Germany. 2014.

[4] P. Sharma Campos, E. Jantunen, D. Baglee, and L. Fumagalli, "-e challenges of cybersecurity frameworks to protect data required for the development of advanced maintenance" in *Procedia CIRP, vol. 47*. p. 227. 2016.

[5] C. Burnikel and J. Ziegler, "Fast recursive division," *Max-Planck-Institut fur Informatik Research Report MPI-I-98-1-022*, Max–Planck–Institut fur Informatik, Saarbrucken, Germany, 1998.

[6] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Manual for using homomorphic encryption for bioinformatics" in *Proceedings of the IEEE*, vol. 105, no. 3. 2017.

[7] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphicencryption for arithmetic of approximate numbers," in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIA-CRYPT'17)*, pp. 409–437. Hong Kong, China. December 2017.

CHAPTER 6

New Features in C++20

In C++20, new features have been introduced. These new features will impact how the code is written by increasing the elegance, reliability, and security metrics. Cryptographers and information security engineers will find in these features new challenges and interesting concepts that can be incorporated in the source code of their implementations. In this chapter, we will cover the most important features and how they can be used in order to improve code for cryptographic applications.

The new features of C++20 are divided in three categories: *language features, library features,* and *headers features.* We will examine most of the features from a cryptography and cryptanalysis point of view and we will present how they can be used in the implementation of cryptographic algorithms.

Feature Testing

According to the standard proposals, C++20 offers new features for testing by defining a set of macros specific to the preprocessor that corresponds to the C++ language. In the following sections, we will discuss two important features that can be used in the process of cryptographic algorithm implementation. These two features will help you to take control of the memory and how the cryptographic structures and operations are represented and designed.

carries_dependency

Although `carries_dependency` was introduced in C++11, it's worth a mention because it is useful in secure applications. In cryptography, working with memory can be a hard task in each of the steps of the algorithm. With help of `carries_dependency` we can skip the limitations and guards for memory [1].

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_6

Starting with C++20, the `std::memory_order` enumeration class structure is as follows [1]:

```
enum class memory_order: {

    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed =
                                        memory_order::relaxed;
inline constexpr memory_order memory_order_consume =
                                        memory_order::consume;
inline constexpr memory_order memory_order_acquire =
                                        memory_order::acquire;
inline constexpr memory_order memory_order_release =
                                        memory_order::release;
inline constexpr memory_order memory_order_acq_rel =
                                        memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst =
                                        memory_order::seq_cst;
```

`std::memory_order` specifies how the memory is accessed, based on the most known atomic operations.

As an example, let's consider a `memory_order::relaxed` inline instruction and look at how it can be used. As a warning, you should use it with caution because, according to the official documentation, "there are no synchronization or ordering constraints imposed on other reads or writes; only this operation's atomicity is guaranteed."[1] This is very useful for complex cryptography algorithms and security schemes, such as searchable encryption, homomorphic encryption, or cryptosystems based on elliptic-curve mathematics. In [1] there is a set of examples meant to explain how to work with the memory using `memory_order`. The first parameter of `fetch_add` represents the type of value accepted. This is essential if you don't need any synchronization between reading or writing operations.

```
std::atomic<int> counter = {0}
counter.fetch_add(1, std::memory_order_relaxed)
```

[1] https://en.cppreference.com/w/cpp/atomic/memory_order

The constants `memory_order_relaxed`, `memory_order_consume`, and `memory_order_release` are the ones most often used for the memory workload done by cryptographic implementations. It is very important to understand their meaning and how to work with them.

- `memory_order_relaxed`: It means that there is no synchronization for reading or writing operations.
- `memory_order_consume`: Cryptographic operations that use this constant are making a consume operation for the referred memory location.
- `memory_order_release`: No reading or writing within the current thread are not allowed for reordering.

In the next example, you can see how `std::memory_order_release` is used when you want to make a release on the memory for a variable. Note that in the example the `cryptoKey` variable represent a general case of storing a cryptographic key (public or private key). In this case, at any point of algorithm the key can be changed once it is released.

```
std::string* cryptoKey = new std::string("passkey");
ptr.store(p, std::memory_order_release)
```

no_unique_address

The data member that is preceded by `no_unique_address` does not need to have an address that is different from all other data members of its class [2].

In the following example, you can see that there is an empty class, `CryptographyOps`. Following this class are two more classes, `Encryption` and `Decryption`. In the `Encryption` class, you can see `Empty CryptographyOps`, which means that the size must be at least 1 even if the type is an empty class, such as in this case. This is necessary in order to guarantee that the distinct objects and their addresses with the same type are always different. In the second class, `Decryption`, you add `[[no_unique_address]]` in front of the `CryptographyOps` class, which will make sure that a distinct address will not be provided.

```
struct CryptographyOps {}; // empty class

struct Encryption {
    int i;
    CryptographyOps CryptoOps;
};

struct Decryption {
    int i;
    [[no_unique_address]] CryptographyOps CryptoOps;
};
```

New Headers in C++20

In this section, we will cover the most important headers in C++20. They are meant to help to improve the development process in the field of cryptography.

<concepts> Header

This new C++20 concept is a part of the concepts library. The concepts library contains fundamental concepts that can be used with the goal of performing compile-time validation. The validation is done with respect for the arguments of the templates and classes. As a second purpose, they realize the dispatching process on the properties of the types.

The header is structured in different concept categories, such as concepts related to the language core, concepts for comparisons, concepts related to the objects and how they are structured, and callable concepts.

Core Language Concepts

Most of the concepts related to the core language are related to the objects and types. In cryptography, these are concepts such as `std::floating_point`, `std::destructible`, `std::integral`, `std::signed_integral`, and `std::unsigned_integral`. In the following sections, we'll examine some examples to show how to work with these concepts in cryptography. Most of these concepts can be used in a cryptography implementation to check types and to make sure that you are on the same page with the types of classes, objects, structures, variables etc.

std::floating_point

Let's consider the following class declaration:

```
//** the class is a general class for working with cryptographic
//** operations related to floating numbers.
class FloatingClassExample {};
```

If we invoke

```
std::cout << std::floating_point<FloatingClassExample>::value;
```

the output will be false. But if we have

```
std::cout << std::floating_point<float>::value;
```

the output will be true.

std::destructible

In cryptography, once we reach the end of an operation (e.g. encryption or decryption) it is recommended that all functions, structures, classes, objects, and types be destroyed.

To achieve the destruction process using `std::destructible`, we just need to pass the type to `destructible<>` as follows:

```
class AESClass {};
std::cout << std::destructible<AESClass>::value;
```

The output of this will be true.

std::integral, std::signed_integral, std::unsigned_integral

The concepts are used in the same way as the examples listed above. Let's consider the following type tests:

```
std::cout << integral<AESClass>::value;
std::cout << integral<int>::value;
std::cout << integral<float>::value;
```

The output will be false, true, and false. For `signed_integral` and `unsigned_integral`, it's the same.

Comparison Concepts

In C++20, there are four comparison concepts, `equality_comparable`, `equality_comparable_with`, `totally_ordered`, and `totally_ordered_with` [4]. They work with comparison operators such as `==, !=, <, >, <=, >=` over the specified type. There is a slight difference between the comparison concepts, despite the fact that both work with the mentioned operators. The `concept totally_ordered` is focused on the fact that the results of the concepts are yielded with a strict total order on the type [3] and `totally_ordered_with<operand1, operand2>` is focused on mixed operands, which yield the results with a strict total order.

Object Concepts

The object concepts refer to the main operations with objects, such as *moving*, *swapping*, and *copying*. There are four object concepts that work with operations on objects: *movable*, *copyable*, *semiregular*, and *regular*. These concepts are quite useful in cryptography, especially when dealing with complex cryptographic objects that must be moved, copied, or swapped during their execution of interchange process between different software infrastructures, operating systems, and software applications. In this way, we can increase the security of the interoperability of the applications.

As an example, let's have a quick look over the following:

```
template <class AESCrypto>
concept_movable =
                std::is_object_v<AESCrypto> &&
                std::move_constructible<AESCrypto> &&
                std::assignable_from<AESCrypto&, AESCrypto> &&
                std::swappable<AESCrypto>;
```

Callable Concepts

Firstly, let's clear the fog around the *callable* term and define the callable concept as being "something" that can be called, such as a function. Usually, it's `object()` or `object(arguments)`. What makes an object callable is the overload of the `operator()` function.

With C++20, the following concepts are covered: `invocable/regular_invocable`, `predicate`, `relation`, `equivalence_relation`, and `strict_weak_order`. In cryptography, we deal more with `predicate` and `relation`. Once these two concepts are fully understood, the rest of the concepts are easy.

With the `predicate` concept, the arguments provided will produce a Boolean result. Compared to the `relation` concept, we have the following example:

```
template<class AESCrypto, class Encryption, class Decryption>
concept relation=
       std::predicate<AESCrypto, Encryption, Decryption> &&
       std::predicate<AESCrypto, Decryption, Decryption> &&
       std::predicate<AESCrypto, Encryption, Decryption> &&
       std::predicate<AESCrypto, Decryption, Encryption>;
```

As you can observe, according to `relation<AESCrypto, Encryption, Decryption>` we have the fact the `AESCrypto` will define a binary relation in accordance with the set of expressions characterized by the type and value that are encoded by `Encryption` or `Decryption`.

<compare> Header

The `<compare>` header [5] deals with comparing operators, and it is a component of the general utility library.

Starting with C++20, we have access through this header to powerful *concepts* (`three_way_comparable` and `three_way_comparable_with`), to *classes* (`partial_ordering`, `weak_ordering`, and `strong_ordering`), and to *customized point objects* (`strong_order`, `weak_order`, and `partial_order`).

As you saw in the "Comparison Concepts" section, the mechanisms are the same. If you are working with complex applications that are using threads and the data used and transferred on those threads are sensitive, the sender and receiver could verify through comparison functions and concepts if the integrity of the data was sent and received properly. These functions extend the power of cryptography algorithms and give you a free hand to create complex mechanisms for comparing and working with sensitive data at both ends.

<format> Header

In cryptography, we deal with rules. We have rules for numbers representation, rules for strings, rules for characters, mathematical rules on expressions, and so on.

The <format> header extends its capabilities in C++20 by bringing new insights regarding how the formatting rules are defined and implemented. Although at the moment of writing this book, the format header is not yet included in the C++ standard library, it worth keeping it in mind for cryptography applications. Updates for the C++ libraries and features can be found at [6]

As an example, consider the program in Listing 6-1.

Listing 6-1. The <format> Header

```
#include <format>
#include <iostream>

//** let's define a wrapper for class AESCryptography
template<class AESCryptography>
struct Encryption {
    AESCryptography value;
};

template<class AESCryptography, class CharAESCryptography>

struct
std::formatter<Encryption<AESCryptography>,
        CharAESCryptography>:
std::formatter<AESCryptography, CharAESCryptography>
{
    template<class FormatContext>
    auto format(Encryption<AESCryptography>
         encAESCrypto, FormatContext& theFormatContext)
    {
        return std::formatter<AESCryptography,
          CharAESCryptography>::format
             (encAESCrypto.value, theFormatContext);
    }
};
```

```
int main()
{
    Encryption<int> encrypted = { 32 };
    std::cout << std::format("{:#x}", encrypted);
}
```

Conclusion

In this chapter, we discussed the most important features in C++20 and how they can help professionals in the field of applied cryptography. We focused on the vital points of the implementation processes of a cryptographic algorithm. The features included in this chapter were designed to cover the main cryptographic operations with respect to memory and type comparisons:

- Features for testing the preprocessors for the C++20 language
- Memory insights and how you can manipulate them more professionally and elegantly
- The `carries_dependency` and `no_unique_address` concepts, which are very useful for working with memory
- The `<concepts>`, `<compare>`, and `<format>` headers and their strongest classes and functions, which can be used for dealing with cryptographic mechanisms such as key generation operations, encryption and decryption functions, and testing and validating types

References

[1] `std::memory_order`. Available online: https://en.cppreference.com/w/cpp/atomic/memory_order.

[2] C++20 – `no_unique_address`. Available online: https://en.cppreference.com/w/cpp/language/attributes/no_unique_address.

[3] Total Order. Available online: https://en.wikipedia.org/wiki/Total_order#Strict_total_order.

[4] std::totally_ordered, std::totally_ordered_with. Available online: https://en.cppreference.com/w/cpp/concepts/totally_ordered.

[5] Standard Library header <compare>, Concepts. Available online: https://en.cppreference.com/w/cpp/utility/compare/three_way_comparable.

[6] Standard Library header <format>. Available online: https://en.cppreference.com/w/cpp/header/format.

[7] C++ compiler support. Available online: https://en.cppreference.com/w/cpp/compiler_support.

CHAPTER 7

Secure Coding Guidelines

Vulnerabilities in software applications often result in high costs. Some organizations pay more than $500,000 per security incident. To eliminate vulnerabilities from software applications, developers should focus on secure coding and thus avoid deploying any vulnerabilities in the production phase.

Writing secure source code is a difficult task. It is very important to understand the implications of the code that is being written and to have a checklist with the "things" that need to be checked. The checklist will help the developers pursue a fast verification of their code for well-known security problems. Usually, the verification is done by a security team and not by the software developers or engineers. A software developer cannot be objective with their own code.

The idea of a checklist should start from the following concept: verify the source code that will process data outside of its domain and take into consideration user input, the network communication, the process of the binary files, receiving output from database management systems or servers, etc.

When you work with a software application (desktop, web, or mobile), the idea that the application is secure because it was developed by a well-known company is just a myth. Don't just trust and go on this path because most companies will end up spending a huge amount of the budget on security incidents, maintenance, consultancy, and audit sessions.

There are two environments in which a software application works and its behavior is different in each environment. The software application that is under analysis and the development process within the company represents its circle of trust (at least most companies think in this way, and they enjoy considering their infrastructure very resistant to security attacks). The behavior of the software application in that circle of

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_7

trust represents the most critical environment in which an application can be developed and tested. No developer, IT security officer, or software analyst should hack their own code. This environment is the comfort zone. Once the application leaves that comfort zone and enters the real environment, issues will start to take place. The trust boundary is hard and easy at the same time to be drawn. To create a line between the comfort zone and the real zone is not an easy task, especially if the application is running in a virtualized infrastructure, the cloud, or a big data environment.

In the comfort zone, a security threat is represented by the malicious end user. The malicious end user will aim to compromise the confidentiality and/or the integrity of the software application. One of the interesting concepts proposed is *software obfuscation.*

Secure Coding Checklist

In this section, we will discuss and propose a secure coding checklist (it can be seen as a procedure as well). Table 7-1 shows an example of such a checklist and it can be developed as much as you want. The checklist contains minimal examples of items that can be checked when code is written in C++, no matter which operating system the code runs on. A frequent practice among developers is to suppress warnings, which is not beneficial.

In the "CERT Coding Standards and Rules" section, we will discuss the most important rules and list them in order so you can apply them to your process of developing cryptographic algorithms. Each rule is well explained within the guide.

Table 7-1. *Example of a Secure Coding Checklist*

No. #	Item to be checked	Description	Yes/No	Notes
1	***Compiler warnings***			
Make sure that the compiler will output and a flag will be raised for receiving notifications of the potential errors listed for the following items: ✓ `Wall` ✓ `Wmissing-declarations` ✓ `Wmissing-prototypes` ✓ `Wredundant-decls` ✓ `Wshadow` ✓ `Wstrict-prototypes` ✓ `Wformat=2` For more flags with their definitions and actions, refer to the "GCC Options to Request or Suppress Warnings" section [1]. It is very useful if complex cryptographic algorithms and security schemes are being implemented.			
2	***Allocate enough memory for buffer memory when working with strings.***			
Check the following functions if there is an upper limit for the destination buffer when a copy process is done until `'\0\'` (`NULL`) is met. In order to avoid this situation, the recommendation is to allocate enough memory space for the destination buffer before the data is copied there. ✓ `strcpy()` ✓ `strcat()` ✓ `sprint()` ✓ `scanf()` ✓ `gets()`				

(*continued*)

Table 7-1. (*continued*)

No. #	Item to be checked	Description	Yes/No	Notes
3	***Check for direct breaks of system security.***			

Checking for untrusted input will lead to a direct breach of the application security. With this step, you protect the application against malicious users and attackers exploiting your program using metachars.

- ✓ `system()`
- ✓ `popen()`
- ✓ `fork(2)`
- ✓ `exec(2)`
- ✓ `s_popen()`
- ✓ `HXproc_* [2]`

4	***Check for the wrong size of parameters and getting unexpected results.***			

When complex programs are written, such as the implementation of SHA-256 from Chapter 2, Listing 2-1, assigning a wrong size to one of the parameters or doing a wrong arithmetic operation can cause a serious pitfall and a fix should immediately be provided. Make sure that the size allocated for the parameters is the same size on the destination side. As a best practice, especially in implementing cryptography algorithms, it is better to work with `size_t` type. Be type-safe and don't create overflows.

- ✓ `strncpy()`
- ✓ `strncat()`
- ✓ `snprintf()`

(*continued*)

Table 7-1. (*continued*)

No. #	Item to be checked	Description	Yes/No	Notes
5	***Check if too much memory is allocated.***			

Allocating too much memory and external parameters represents a certain part of the size. If so, you are dealing with a wrong memory allocation and you will experience a *denial-of-service.* To avoid this from happening, it is better to follow the below criteria:

- ✓ `malloc(), calloc(), alloca()`
- ✓ No integer overflows
- ✓ Avoid arithmetical issues
- ✓ Verification for any possible operation with untrusted integer that could lead for an integer overlow.

6 ***Avoid wrong casts.***

Avoid coding like below. The compiler will think that `malloc` will return an int, which is totally incorrect. It will create a bug that can easily be exploited by hackers.

```
  char *a = malloc(10) – bad cast
class BaseClass {};
class DerivedClass: public BaseClass {};

BaseClass b; BaseClass* pb;
DerivedClass d; DerivedClass* pd;

//good cast
pb = dynamic_cast<BaseClass*>(&d);

//bad cast
pd = dynamic_cast<DerivedClass*>(&b);
```

(*continued*)

Table 7-1. (*continued*)

No. #	Item to be checked	Description	Yes/No	Notes
7	***Avoid variable parameter lists.***			

When you are implementing security schemes based on strings, you may experience a new type of problem, which security analysts or ethical hackers enjoy playing with when performing tests. A simple test that is commonly used by ethical hackers to check untrusted data is to check if a function allows a variable as a list of parameters or arguments, such as `printf()`. The untrusted data (created by an ethical hacker) is directly used as a string format and not as an argument. Follow the below logic for any similar situations:

- ✓ `Wrong way: snprintf(buffer, sizeof(buffer), the_input_of_the_user)`
- ✓ `Right way: snprintf(buffer, sizeof(buffer), "%s", the_input_of_the_user)`

8	***Operations with files***			

When handling files during cryptographic operations, try to use `mkstemp()`.

9	***File permissions***			

Not everyone should have the ability to read or write from or to a file. In order to create files with the correct permissions, make a habit of using `unmask()`.

- ✓ `At the beginning of the file use unmask(077).`

CERT Coding Standards

The CERT C++ Coding Standard was developed only for versions of the C++ programming language defined by the ISO/IEC 14882-2014 standard.

The coding standard is very well organized and it follows a certain structure: identifiers, noncompliant code examples and compliant solutions, exceptions, risk assessment, automated detection, related vulnerabilities, and related guidelines [7].

We will examine each item of the structure and we will explain the main objective and purpose.

Identifiers

Each identifier has three parts:

- A three-letter mnemonic that represents the section within the standard
- A numeric value of two digits in the range of 00 to 99
- The language that is associated with it, which is represented as a suffix (-CPP, -C, -J, -PL)
  - –CPP: SEI CERT C++ Coding Standard [7]
  - –C: SEI CERT C Coding Standard [8]
  - –J: SEI CERT Oracle Coding Standard for Java [9]
  - –PL: SEI CERT Perl Coding Standard [10]

The three-letter mnemonic is used to group related coding practices and to point out which category a related coding belongs to.

Noncompliant Code Examples and Compliant Solutions

The examples of noncompliant code show the code that violates the guideline. It is very important to keep in mind that they are only examples. The removing process of all appearances of the example does not mean that the code we are analyzing is compliant with the SEI CERT standard.

Exceptions

Exceptions are of an informative character and are not required to be followed. Any of the rules can have a set of exceptions which provide details about the circumstances in which the guideline is not necessary to be followed for ensuring the safety, security, or reliability of the software.

As for any type of exception, it doesn't matter the programming language: the principle is the same. It's necessary to pay extra attention to the exceptions and to catch any possible exception and to learn from it. Don't ignore them, and don't think that a programming language is perfect and doesn't have any bugs or certain doors that can be exploited.

Risk Assessment

Each guideline from the CERT C++ Coding Standard has a risk assessment section. The purpose of the risk assessment section is to provide software developers with the potential consequences of not following or addressing a specific rule or recommendation. The risk assessment looks like a metric and its main purpose is to help the remediation process of the software applications and complex projects.

Each rule and recommendation has a *priority*. In order to assign a priority, it is recommended to understand IEC 60812 [11]. The priority is evaluated and assigned using a metric that is characterized by three types of analysis: failure mode, effects, and criticality. Each rule also has a value that is assigned on a scale between 1 and 3, such as severity, likelihood, and remediation cost (see Table 7-2).

Table 7-2. *Assigning Values for Each Rule [7]*

Severity: What are the consequences if the rule is ignored?		
Value	**Meaning**	**Examples of different vulnerabilities**
1	Low	Denial-of-service attack, unexpected termination
2	Medium	Information disclosure without any intention will lead to the violation of the data integrity.
3	High	Running code randomly

Likelihood: Statistically speaking, what is the probability of a flaw being introduced in the code by avoiding and ignoring the rule specifications and leading to a vulnerability that could be exploited by a malicious user?

Value	**Definition**
1	Unlikely
2	Probable
3	Likely

Remediation cost: What are the costs to follow and comply with the rule?

Value	**Definition**	**Detection**	**Correction**
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

For each of the rules the values are multiplied together. The metric in Table 7-3 gives you a measure that can be useful for prioritizing the rules within the application. The values are from 1 to 27. From all 27 values, only 10 different values occur and are available in most of the cases: 1, 2, 3, 4, 6, 8, 9, 12, 18, and 27. Table 7-3 lists the possible interpretations and meanings of the priorities and levels.

Table 7-3. *Levels and Priorities [7]*

Level	Priorities	Possible Interpretation
L1	12, 18, 27	High severity, likely, inexpensive to fix
L2	6, 8, 9	Medium severity, portable, medium cost to fix
L3	1, 2, 3, 4	Low severity, unlikely, expensive to repair

Automated Detection

The rules and recommendations have sections that describe the automated detection process. The mentioned sections have a set of tools which can be used as analyzers to help automatically diagnose any violations. The Secure Coding Validation Suite [12] can be used to perform tests on the ability of analyzers to provide a diagnosis on a violation of the rules specified with ISO/IEC TS 17961:2013 [14], which is related to the rules of the SEI CERT C Coding Standard [13].

Related Guidelines

This section has a special slot when software applications are developed. According to the standard, the "Related Guidelines" section contains links, technical specifications, and guideline collections such as *Information Technology – Programming Languages, Their Environments and System Software Interfaces – C Secure Coding Rules [14]; Information Technology – Programming Languages – Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use [15]; MISRA C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems [16]; and CWE IDs in MITRE's Common Weakness Enumeration (CWE) [17].* [18]

Rules

In the following sections, we will give a short overview of the main rules that strongly apply to the implementation of cryptographic algorithms and security schemes using C++20. It is best to follow these rules. Note that we will examine only six out the 10 rules. All the explanations and examples are provided within the guide [19].

For some rules, there are also rules from the C programming language that apply to C++. The following rules can be used within the procedure explained in the Secure Coding Guidelines in Table 7-1.

The duty of any information security officer, security analyst, or ethical hacker is to improve the code by following such a checklist. Further, the checklist also can be used by developers as a guide when they are developing critical cryptographic algorithms. It is recommended to do a code review on the sections of the algorithm that are quite vulnerable and to make sure that the rules (Rules 1 through 7) are followed as much as possible (see Tables 7-4 through 7-9).

Following these rules will give you as a security analyst or ethical hacker a certain level of trust that the security mechanisms (cryptographic algorithms, security protocols, security schemes, and other cryptographic primitives) have been implemented properly and common vulnerabilities have been eliminated.

Rule 1 - Declarations and Initializations (DCL)

Table 7-4. *Rule 1 – Declarations and Initializations [19]*

Rule	Title
DCL50-CPP	Do not define a C-style variadic function.
DCL51-CPP	Do not declare or define a reserved identifier.
DCL52-CPP	Never qualify a reference type with const or volatile.
DCL53-CPP	Do not write syntactically ambiguous declarations.
DCL54-CPP	Overload allocation and deallocation functions as a pair in the same scope.
DCL55-CPP	Avoid information leakage when passing a class object across a trust boundary.
DCL56-CPP	Avoid cycles during initialization of static objects.
DCL57-CPP	Do not let exceptions escape from destructors or deallocation functions.

(*continued*)

Table 7-4. (*continued*)

Rule	Title
DCL58-CPP	Do not modify the standard namespaces.
DCL59-CPP	Do not define an unnamed namespace in a header file.
DCL60-CPP	Obey the one-definition rule.
DCL30-C	Declare objects with appropriate storage durations.
DCL39-C	Avoid information leakage when passing a structure across a trust boundary.
DCL40-C	Do not create incompatible declarations of the same function or object.

Rule 2 - Expressions (EXP)

Table 7-5. *Rule 2 – Expressions [19]*

Rule	Title
EXP50-CPP	Do not depend on the order of evaluation for side effects.
EXP51-CPP	Do not delete an array through a pointer of the incorrect type.
EXP52-CPP	Do not rely on side effects in unevaluated operands.
EXP53-CPP	Do not read uninitialized memory.
EXP54-CPP	Do not access an object outside of its lifetime.
EXP55-CPP	Do not access a cv-qualified object through a cv-unqualified type.
EXP56-CPP	Do not call a function with a mismatched language linkage.
EXP57-CPP	Do not cast or delete pointers to incomplete classes.
EXP58-CPP	Pass an object of the correct type to `va_start`.
EXP59-CPP	Use `offsetof()` on valid types and members.
EXP60-CPP	Do not pass a nonstandard-layout type object across execution boundaries.
EXP61-CPP	A lambda object must not outlive any of its reference captured objects.
EXP62-CPP	Do not access the bits of an object representation that are not part of the object's value representation.
EXP63-CPP	Do not rely on the value of a moved-from object.

Rule 3 - Integers (INT)

Table 7-6. *Rule 3 – Integers [19]*

Rule	Title
INT50-CPP	Do not cast to an out-of-range enumeration value.
INT30-C	Ensure that unsigned integer operations do not wrap.
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data.
INT32-C	Ensure that operations on signed integers do not result in overflow.
INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors.
INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.
INT35-C	Do not call a function with a mismatched language linkage.
INT36-C	Converting a pointer to integer or integer to pointer

Rule 5 - Characters and Strings (STR)

Table 7-7. *Rule 5 – Characters and Strings [19]*

Rule	Title
STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator.
STR51-CPP	Do not attempt to create a `std::string` from a null pointer.
STR52-CPP	Use valid references, pointers, and iterators to reference elements of a `basic_string`.
STR53-CPP	Range check element access.
STR30-C	Do not attempt to modify string literals.
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator.

(*continued*)

Table 7-7. (*continued*)

Rule	Title
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string.
STR34-C	Cast characters to unsigned char before converting to larger integer sizes.
STR37-C	Arguments to character-handling functions must be representable as an unsigned char.
STR38-C	Do not confuse narrow and wide character strings and functions.

Rule 6 - Memory Management (MEM)

Table 7-8. *Rule 6 – Memory Management [19]*

Rule	Title
MEM50-CPP	Do not access freed memory.
MEM51-CPP	Properly deallocate dynamically allocated resources.
MEM52-CPP	Detect and handle memory allocation errors.
MEM53-CPP	Explicitly construct and destruct objects when manually managing object lifetime.
MEM54-CPP	Provide placement `new` with properly aligned pointers to sufficient storage capacity.
MEM55-CPP	Honor replacement dynamic storage management requirements.
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer.
MEM57-CPP	Avoid using default operator `new` for over-aligned types.
MEM30-C	Do not access freed memory.
MEM31-C	Free dynamically allocated memory when no longer needed.
MEM34-C	Only free memory allocated dynamically.
MEM35-C	Allocate sufficient memory for an object.
MEM36-C	Do not modify the alignment of objects by calling `realloc()`.

Rule 7 - Input/Output (FIO)

Table 7-9. *Rule 7 – Input/Output [19]*

Rule	Title
FIO50-CPP	Do not alternately input and output from a file stream without an intervening positioning call.
FIO51-CPP	Close files when they are no longer needed.
FIO30-C	Exclude user input from format strings.
FIO32-C	Do not perform operations on devices that are only appropriate for files.
FIO34-C	Distinguish between characters read from a file and EOF or WEOF.
FIO37-C	Do not assume that `fgets()` or `fgetws()` returns a nonempty string when successful.
FIO38-C	Do not copy a FILE object.
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call.
FIO40-C	Reset strings on `fgets()` or `fgetws()` failure.
FIO41-C	Do not call `getc()`, `putc()`, `getwc()`, or `putwc()` with a stream argument that has side effects.
FIO42-C	Close files when they are no longer needed.
FIO44-C	Only use values for `fsetpos()` that are returned from `fgetpos()`.
FIO45-C	Avoid TOCTOU race conditions while accessing files.
FIO46-C	Do not access a closed file.
FIO47-C	Use valid format strings.

Conclusion

In this chapter, you learned about rules and recommendations. You pursued a journey through the most important security aspects that need to be taken into consideration in the process of developing cryptographic algorithms and security schemes.

It is very important to understand the difference between a rule and a recommendation. The general idea is that a *rule* has to follow a specific amount of criteria compared to a *recommendation,* which is a suggestion for improving code quality.

You now have enough knowledge to perform a security analysis of the source code, create a secure coding checklist, filter those aspects that are vital for your application, and instruct the developers on how to proceed when they are implementing cryptographic algorithms and written related source code.

References

[1] GCC Options to Request or Suppress Warnings. Available online: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options.

[2] HXprox_*, libHX – Get Things Done. Available online: http://libhx.sourceforge.net/.

[3] [ISO/IEC TR 24772:2013] ISO/IEC. Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use. TR 24772-2013. ISO. March 2013.

[4] [ISO/IEC TS 17961:2012] ISO/IEC TS 17961. Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules. ISO. 2012.

[5] [ISO/IEC 14882-2014] ISO/IEC 14882-2014. *Programming Languages — C++, Fourth Edition.* 2014.

[6] A. Ballman, SEI CERT C++ Coding Standard (2016 *e*dition) 435. 2016.

[7] SEI CERT C++ Coding Standard: Available online: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682 (accessed 4.9.20).

[8] SEI CERT C Coding Standard. Available online: https://wiki.sei.cmu.edu/confluence/display/c (accessed 4.9.20).

[9] SEI CERT Oracle Coding Standard for Java. Available online: https://wiki.sei.cmu.edu/confluence/display/java (accessed 4.9.20).

[10] SEI CERT Perl Coding Standard. Available online: https://wiki.sei.cmu.edu/confluence/display/perl (accessed 4.9.20).

[11] [IEC 60812 2006] IEC (International Electrotechnical Commission). *Analysis Techniques for System Reliability—Procedure for Failure Mode and Effects Analysis (FMEA)*, second edition (IEC 60812). Geneva, Switzerland: IEC. 2006.

[12] Secure Coding Validation Suite. Available online: https://github.com/SEI-CERT/scvs.

[13] SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 edition), n.d. 534.

[14] [ISO/IEC TS 17961:2012] ISO/IEC TS 17961. Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules. ISO. 2012.

[15] [ISO/IEC TR 24772:2013] ISO/IEC. Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use. TR 24772-2013. ISO. March 2013.

[16] MISRA Limited. *MISRA C++ 2008 Guidelines for the Use of the C++ Language in Critical Systems*. ISBN 978-906400-03-3 (paperback); ISBN 978-906400-04-0 (PDF). June 2008.

[17] MITRE. Common Weakness Enumeration, Version 1.8. February 2010. Available online: http://cwe.mitre.org/.

[18] How this Coding Standard is Organized. Available online: https://wiki.sei.cmu.edu/confluence/display/cplusplus/How+this+Coding+Standard+Is+Organized.

[19] Rules. Available online: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046322.

CHAPTER 8

Cryptography Libraries in C/C++20

The objective of this chapter is to provide a comprehensive list of C++ libraries that can be used with success with the new features of C++20. This chapter is very useful when you need to access a specific implementation of a particular functionality. You don't need to search different online resources and you have access to source code that you can use and improve upon.

Overview of Cryptography Libraries

Table 8-1 lists the most important cryptography libraries. The selection was based on two metrics: time execution and flexibility, and access to the source code based on open source licenses. Having access to source code is very useful because you can compare your work and algorithms to other algorithms and implementations and thereby improve your work.

Table 8-1. *Main C/C++ Libraries*

Library Title	Developer/Industry	Programming Language	Open Source	References
OpenSSL	OpenSSL Project	C	X	[1][2][3]
Crypto++	Crypto C++ Project	C++	X	[7][8]
Botan	Jack Lloyd	C++	X	[5]
Libcrypt	GnuPC Commmunity	C	X	[9][10]
GnuTLS	Simon Josefsson Nikos Mavrogiannopoulos	C	X	[11][12]
Cryptlib	Peter Gutmann	C	X	[13]

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_8

For each of the libraries we will introduce the best implementations of the cryptographic primitives (such as key generation and exchange, elliptic-curve cryptography, public key cryptography, hash functions, MAC algorithms, block ciphers, etc.).

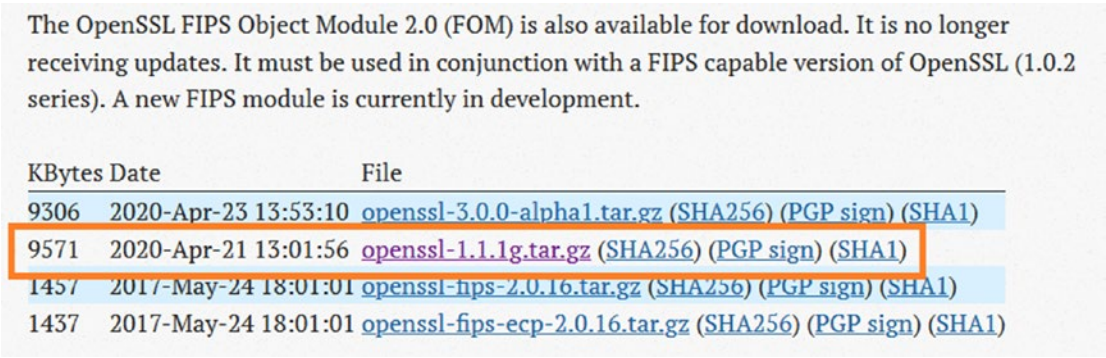
Hash Functions

Table 8-2 shows the cryptography libraries and their features within different hash functions. In *Chapter 2,* we provided a simple and basic implementation of a SHA-256 hash function and you learned how to make an implementation of a hash function from scratch.

Table 8-2. *Existance of Hash Functions Within Cryptography Libraries*

Library Title	MD5	SHA-1	SHA-2	SHA-3	Whirlpool	GOST	BLAKE2
OpenSSL	X	X	X	X	X	X	X
Crypto++	X	X	X	X	X	X	X
Botan	X	X	X	X	X	X	X
Libcrypt	X	X	X	X	X	X	X
GnuTLS	The library represents the implementation of TLS, SSL, and DTLS protocols.						
Cryptlib	X	X	X	X	X	-	-

In this section, we will randomly pick a hash function from a library (e.g. a MD5 implementation from OpenSSL) and we will provide some comments on the implementation. It is very important to mention that the implementation provided for the MD5 hash function is already implemented in OpenSSL and this will be done with respect for the original implementation from [4]. The first thing to do is download the file `openssl-1.1.1g.tar.gz` and extract the content in order to have access to the source code. Once it is extracted, navigate to the `crypto` folder following the path `openssl-1.1.1g\crypto`. In this way, you will have access to the source code files of all the cryptographic algorithms implemented within the library. See Figure 8-1.


The OpenSSL FIPS Object Module 2.0 (FOM) is also available for download. It is no longer receiving updates. It must be used in conjunction with a FIPS capable version of OpenSSL (1.0.2 series). A new FIPS module is currently in development.

KBytes	Date	File
9306	2020-Apr-23 13:53:10	openssl-3.0.0-alpha1.tar.gz (SHA256) (PGP sign) (SHA1)
9571	2020-Apr-21 13:01:56	openssl-1.1.1g.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)
1437	2017-May-24 18:01:01	openssl-fips-ecp-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)

Figure 8-1. *Downloading the openssl-1.1.1g.tar.gz file with source code*

MD5 Hash Function Overview

We picked this example because it is a simple algorithm so it's easy to follow and to understand. The implementation of MD5 contains three files (two C/C++ files and one header file) and an ASM folder with three files written in the PERL language. The PERL files are optimizations for four platforms, 586, x86, x64 and sparc. See Figure 8-2.

Figure 8-2. *Example of a MD5 hash in action for a file*

Public Key Cryptography

Most of the libraries include implementations of different standards of PKCS (Public Key Cryptography Standards) and they are well tested (see Table 8-3).

Table 8-3. *Existance of Public Key Cryptography Protocols Within Cryptography Libraries*

Library Title	PKCS#1	PKCS#5	PKCS#8	PKCS#12	IEEE P1363	ASN.1
OpenSSL	X	X	X	X	-	X
Crypto++	X	X	X	-	X	X
Botan	X	X	X	-	X	X
Libcrypt	X	X	X	X	X	X
Cryptlib	X	X	X	X	-	-

In order to use public key cryptography using OpenSSL, follow the below example to see the workflow. Assume that there are two users, Alice and Bob, who communicate with each other. The communication workflow is as follows.

Step 1. Alice generates a private key, `alicePrivKey.pem` with 2048 bits. See Figure 8-3.

```
openssl genrsa -out alicePrivKey.pem 2048
```

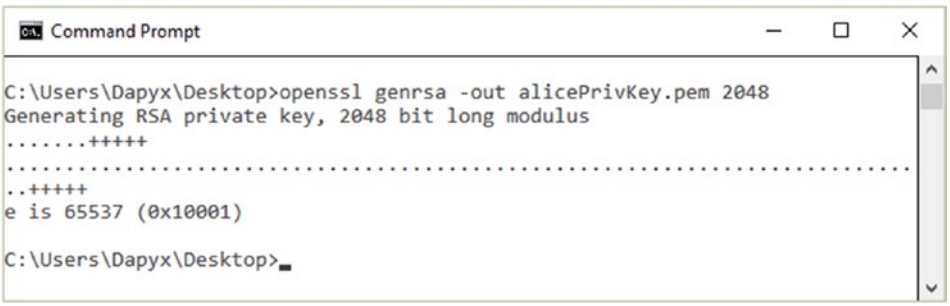


Figure 8-3. *Generating a private key*

Step 2. Alice extracts the public key `alicePublicKey.pem` and sends it to Bob. See Figure 8-4.

```
openssl rsa -pubout -in alicePrivKey.pem -out alicePublicKey.pem
```

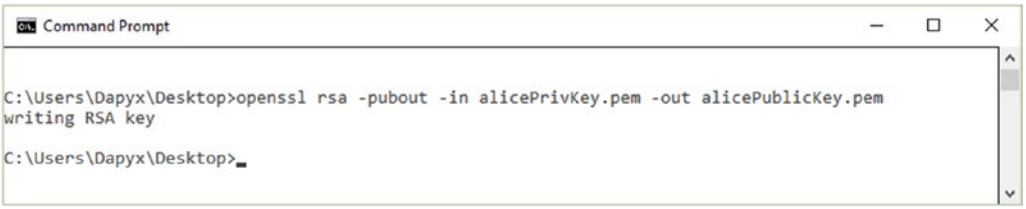


Figure 8-4. *Extracting the public key*

Step 3. Bob encrypts the message and sends `BobMessageToAllice.txt` to Alice.

```
openssl rsautl -encrypt -in cleartext -out encryptedWithAlicePubKey -inkey
alicePublicKey.pem -pubin
```

Step 4: Alice decrypts the message from Bob.

```
openssl rsautl -decrypt -in encryptedWithAlicePubKey -inkey alicePrivKey.pem
```

Elliptic-Curve Cryptography (ECC)

One of the most utilized key exchange protocols based on elliptic curves is ECDH (Elliptic Curve Diffie-Hellman); see Table 8-4. The purpose of this protocol is to set a shared secret key used in the encryption process without needing to send it directly to each of the partners within the communication process.

Table 8-4. *Existance of Elliptic-Curve Cryptography Within Cryptography Libraries*

Library Title	NIST	SECG	ECDSA	ECDH	GOST R 34.10
OpenSSL	X	X	X	X	X
Crypto++	X	X	X	X	-
Botan	X	X	X	X	X
Libcrypt	X	X	X	X	X
Cryptlib	X	X	X	X	-

In order to avoid the mathematic apparatus behind the protocol, we will summarize the workflow of the protocol as follows so that you have a clear overview of the domain parameters that are exchanged between the communication partners (Alice and Bob):

- Alice generates a private key and a public key with the parameters of the domain.
- Bob generates a private key and a public key with the domain parameters set above.
- Both users exchange their public keys.
- Alice computes using the public key of Bob and the shared function is characterized by a shared secret, known as the derived key of B.
- Bob does the same thing with the public key of Alice. The shared function is characterized by a shared secret, known as the derived key of A.
- Alice now uses the derived key of Bob to encrypt the message.
- Bob uses the derived key of Alice to encrypt the message.
- Both users can decrypt the message using their own private key.

Creating ECDH Keys

First, it is quite important to check what OpenSSL support you have on your machine related to ECDH keys. To achieve these primary tasks, run the command `openssl ecparam -list_curves` (see Figure 8-5). The command will provide a full list of curves that you can use. Most of them are implemented properly with respect for their standards. Their implementation in OpenSSL and the recent updates using C++20's new features make them easy to follow.

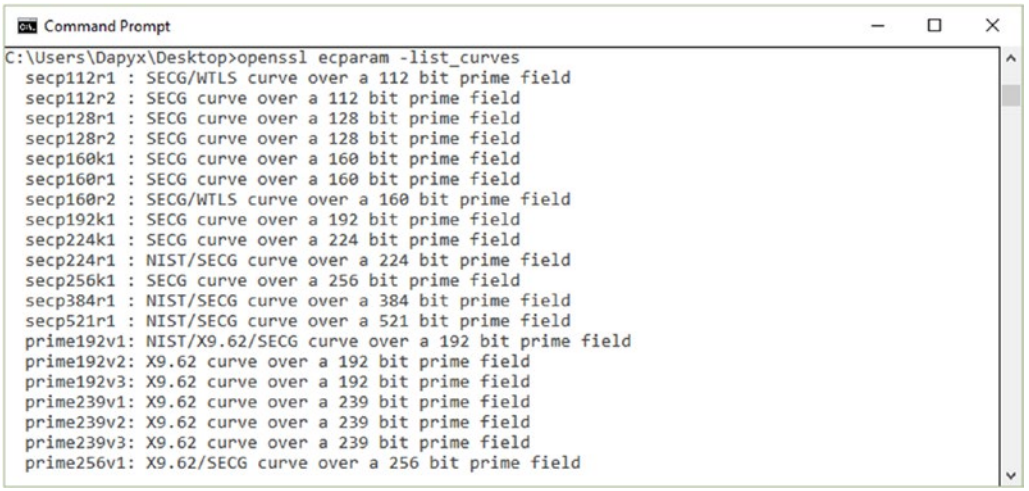


Figure 8-5. *Getting a list of curves*

There is a faster way to create the keypair by using the following command (see Figure 8-6): `openssl ecparam -name prime256v1 -genkey -noout -out key.pem`.

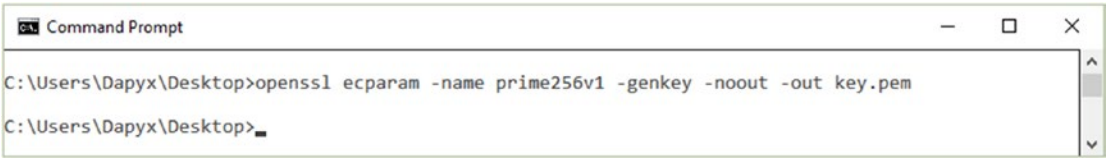


Figure 8-6. *Generating a keypair*

This will output something like

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIKPvG4c8qipI+aZmV3SVnYCZ/QxVfsbJ1CFk4H1uHhFJoAoGCCqGSM49
AwEHoUQDQgAEjQZtqOqt7r8aBkdxVvXxoJC4BdvoPIVBHlsIfw7+GIWmOAJNzFg5
IHbZQF6OhNdWguSl4/Mug24NvGG4PapIhg==
-----END EC PRIVATE KEY-----
```

If you want to see the details of the EC parameter, run the following command: `openssl ec -in key.pem -text -noout`. The command will output something like Figure 8-7.

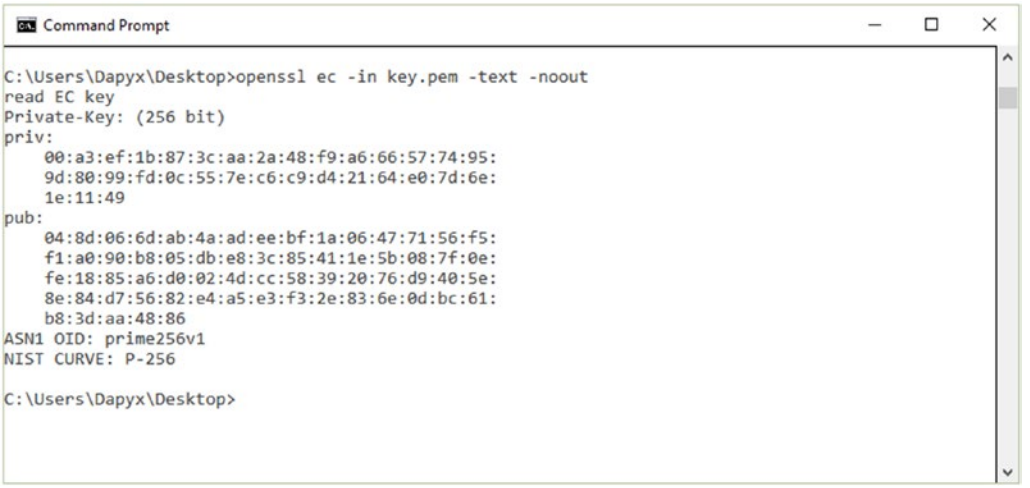


Figure 8-7. *Showing the details of the EC parameter*

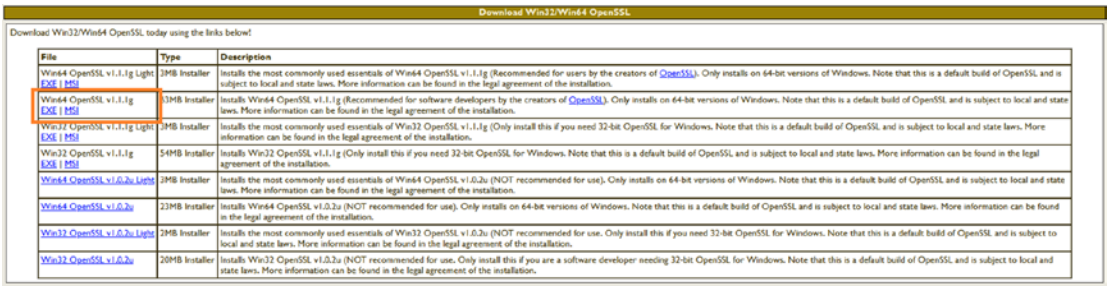
OpenSSL

Configuration and Installing OpenSSL

In order to configure and properly install OpenSSL, depending on the OS platform that is used, refer to the sections below and follow the steps accordingly.

Installing OpenSSL on Windows 32/64

Step 1: Download the binaries for OpenSSL [3]. Download the latest version of OpenSSL Windows Installer by going to `https://slproweb.com/products/Win32OpenSSL.html`. Scroll down until you reach the Download Win32/Win64 OpenSSL option (see Figure 8-8).



Download Win32/Win64 OpenSSL

Download Win32/Win64 OpenSSL today using the links below!

File	Type	Description
Win64 OpenSSL v1.1.1g Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.1.1g (Recommended for users by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.1.1g EXE | MSI	33MB Installer	Installs Win64 OpenSSL v1.1.1g (Recommended for software developers by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1g Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.1.1g (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1g EXE | MSI	54MB Installer	Installs Win32 OpenSSL v1.1.1g (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.0.2u Light	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.0.2u (NOT recommended for use). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.0.2u	23MB Installer	Installs Win64 OpenSSL v1.0.2u (NOT recommended for use). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.0.2u Light	2MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.0.2u (NOT recommended for use. Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.0.2u	20MB Installer	Installs Win32 OpenSSL v1.0.2u (NOT recommended for use. Only install this if you are a software developer needing 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.

Figure 8-8. *Download section of OpenSLL*

Step 2: Double click and run `Win64OpenSSL-1_1_1g.exe` (see Figure 8-9).

Figure 8-9. *Setup of OpenSSL*

Step 3: Accept the license agreement and click Next. See Figure 8-10.

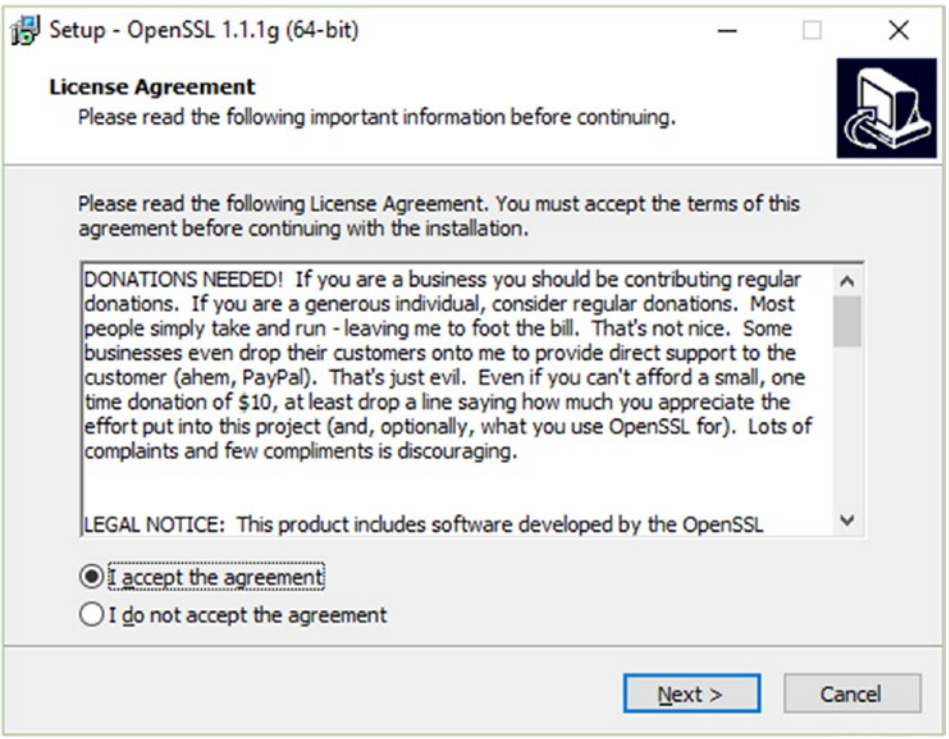


Figure 8-10. *OpenSSL license agreement*

Step 4: Specify the path where it should be installed and click Next. See Figure 8-11.

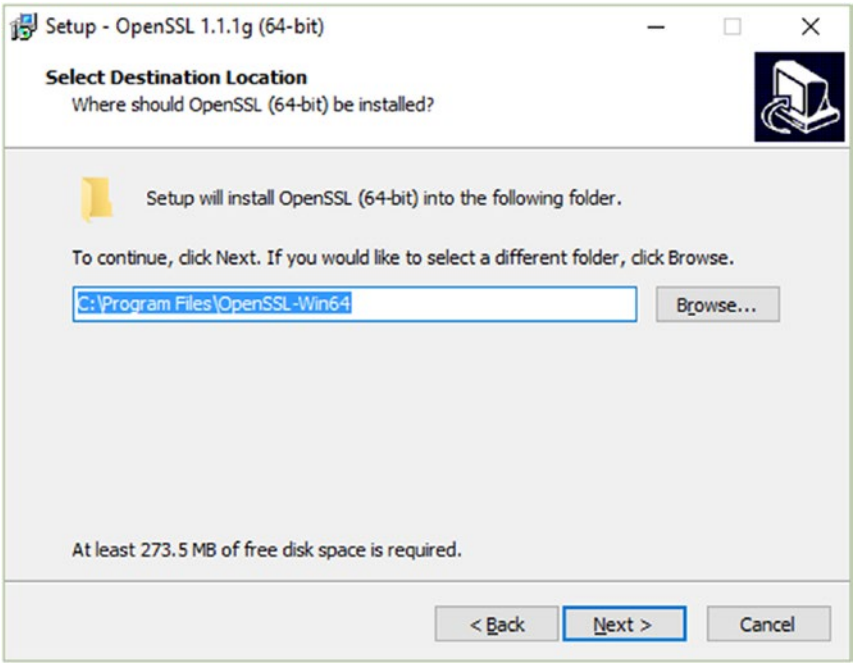


Figure 8-11. *Setting up the path where OpenSSL will be installed*

Step 5: Leave the screen as is and click Next. See Figure 8-12.

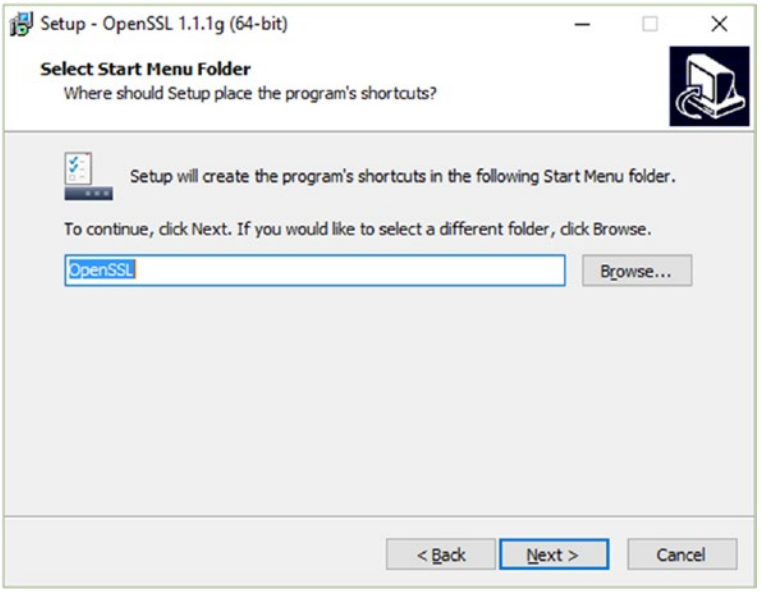


Figure 8-12. *The location of the program shortcuts*

Step 6. Leave everything as is and click Next. See Figure 8-13.

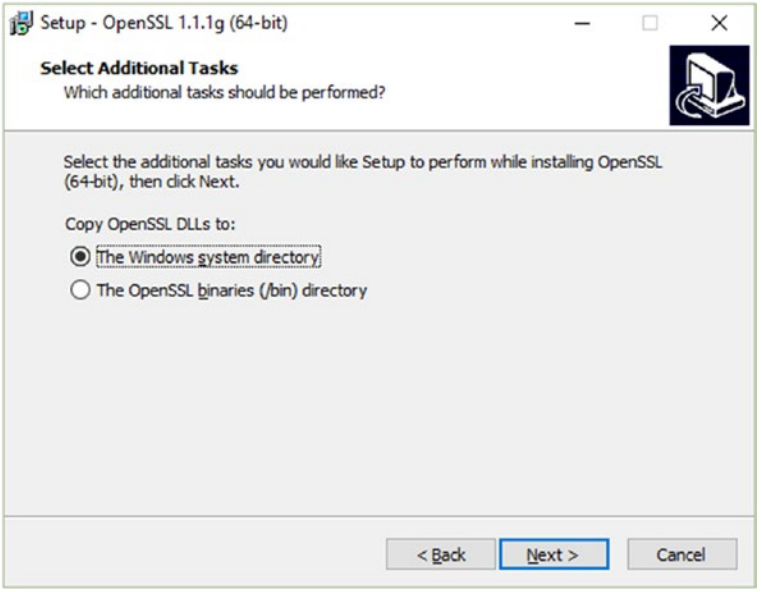


Figure 8-13. *Additional tasks to perform*

Step 7: You're ready for installation so click Install. See Figure 8-14.

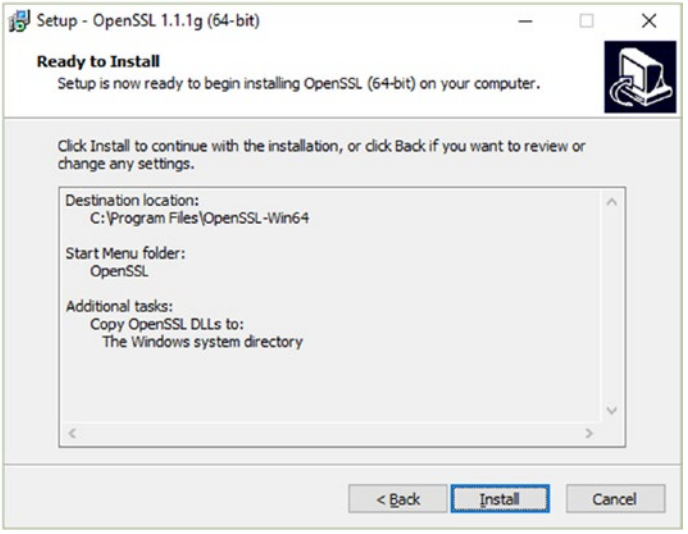


Figure 8-14. *Acknowledgment of the installation process and settings*

Step 8: Installation progress. Remember that if you haven't installed Microsoft Visual C++ Redistributable (x64) it will ask you to install it. See Figure 8-15.

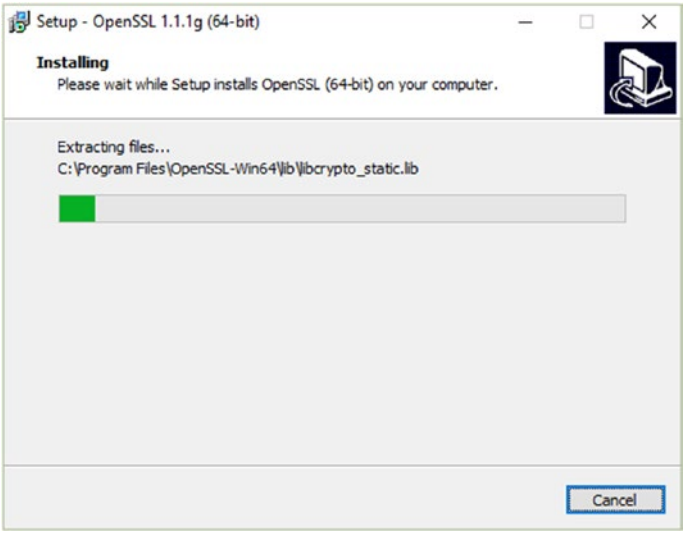


Figure 8-15. *OpenSSL installation progress*

Step 9: Finishing the process of installation. Leave everything as is and click Finish. See Figure 8-16.

Figure 8-16. *Completing the process of installation*

Step 10: Configure and set up the environment variables for OpenSSL.

Step 11: Run the environment variables. Go to System Properties and click Environment Variables. See Figure 8-17.

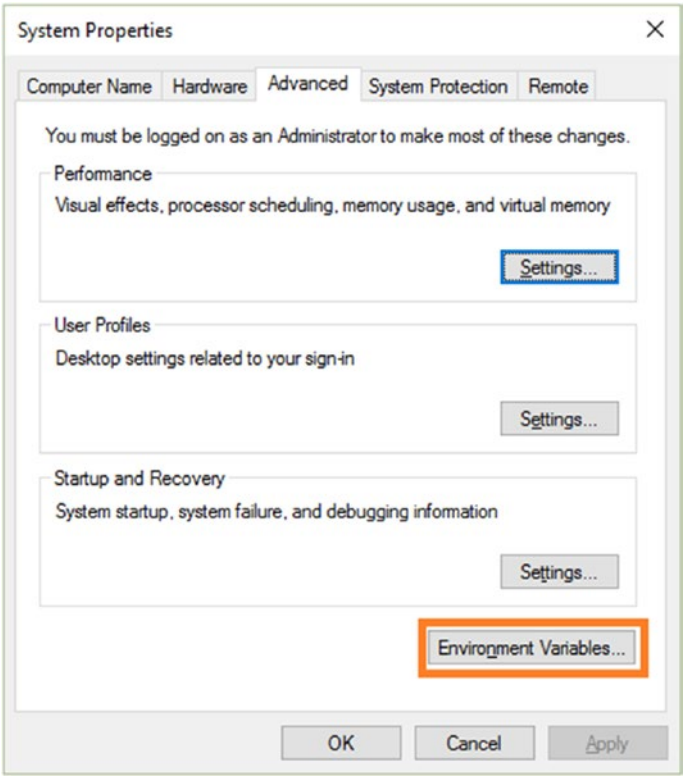


Figure 8-17. *System properties*

Step 11: The environment variable for OpenSSL will be added in System Variables. Click the New button shown in Figure 8-18.

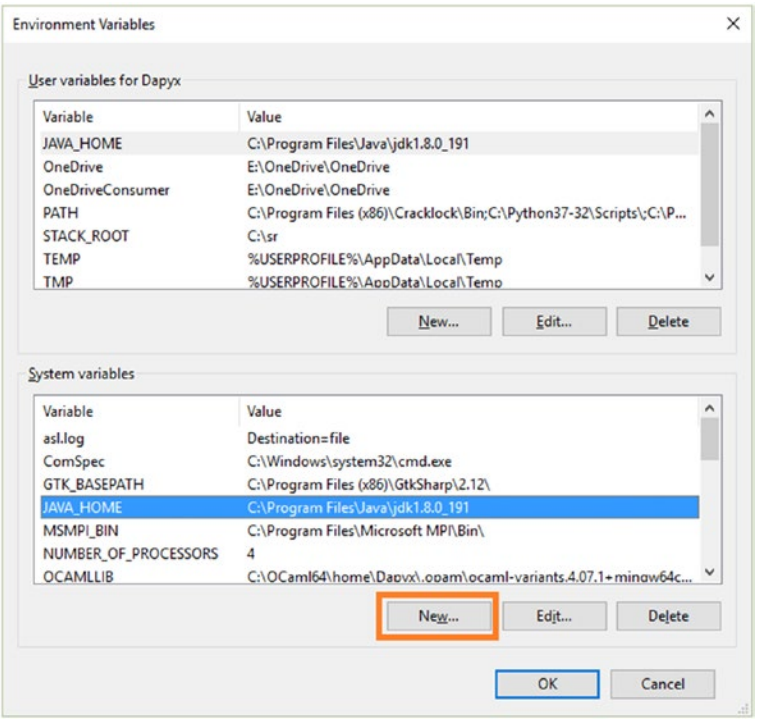


Figure 8-18. *Environment variables*

Step 12: Configure the `OPENSSL_CONF` variable. See Figure 8-19.

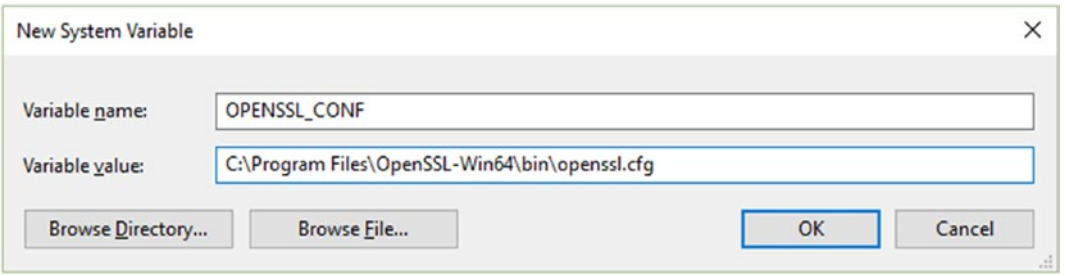


Figure 8-19. *A new system variable*

Step 13: Configure and modify the path variable accordingly. Select from System Variables the Path variable and click Edit. See Figure 8-20.

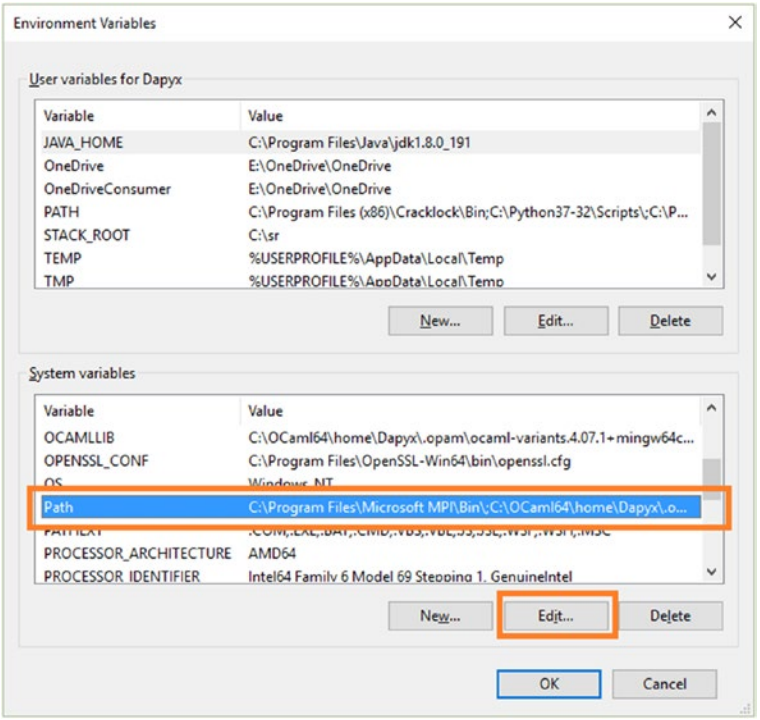


Figure 8-20. *Environment variables ➤ Path variable*

Step 14: In the Edit environment variable section, click New and Browse. See Figure 8-21.

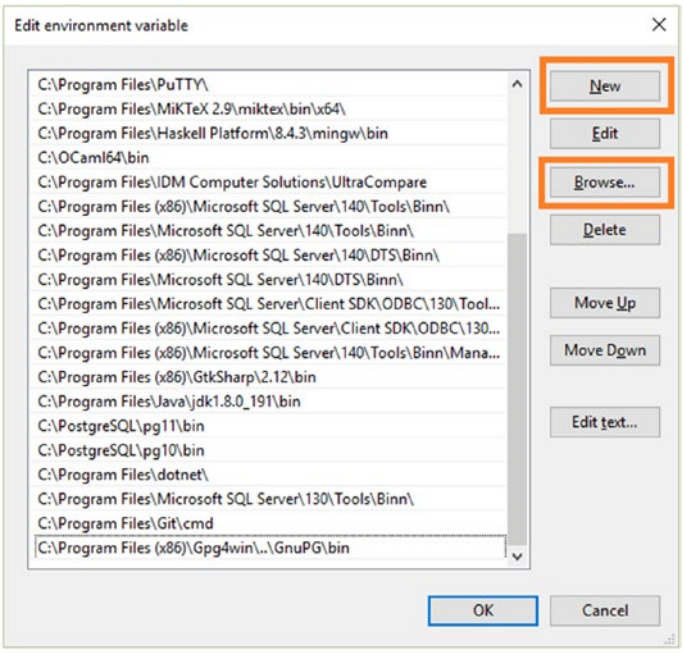


Figure 8-21. *Editing the environment variable*

Step 15: Select the path to the OpenSSL `bin` folder and click Ok. The new path has been added with success. Close everything. If you have the Command window open, close it and reopen it again for the update to be done accordingly. Otherwise, it will not work. See Figure 8-22.

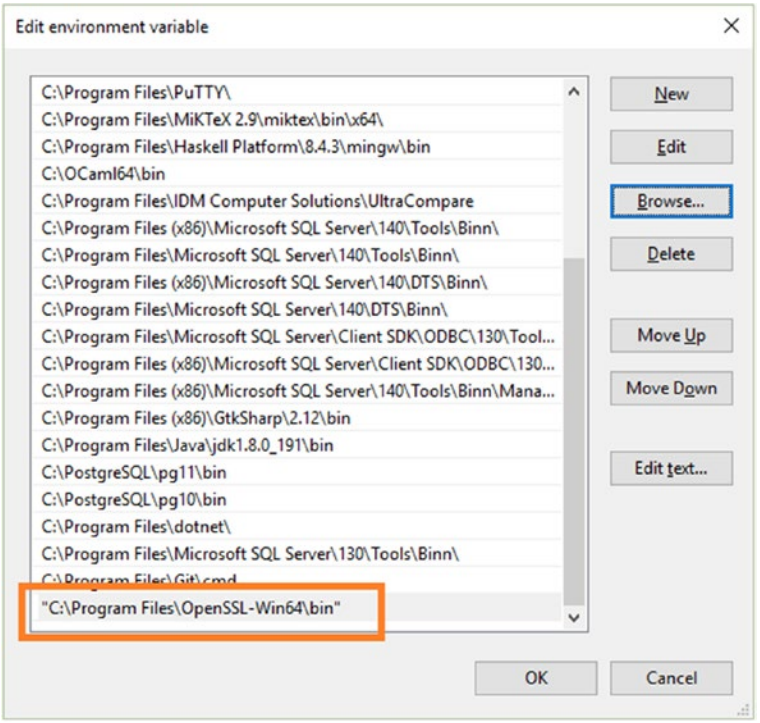


Figure 8-22. *Verifying that the path for OpenSSL has been added*

Step 16: Open Command (`cmd.exe`). Run the `openssl` command. If the OpenSSL ➤ prompter appears in the window, it means that you have the first sign of success. See Figure 8-23.

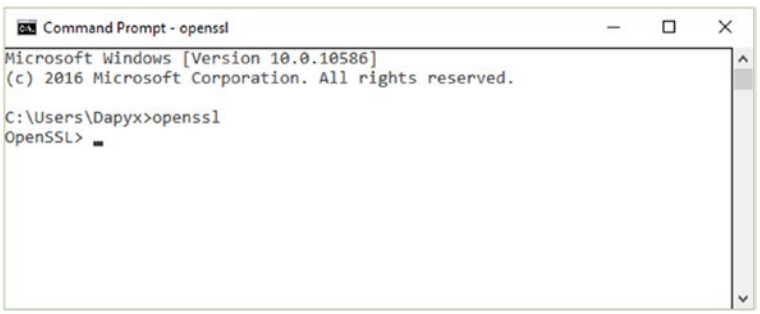


Figure 8-23. *Checking OpenSSL, first step*

Step 17: Run the second command, `version`. Make sure that everything is set properly. If the version and date are returned as shown in Figure 8-24, you can declare yourself successful.

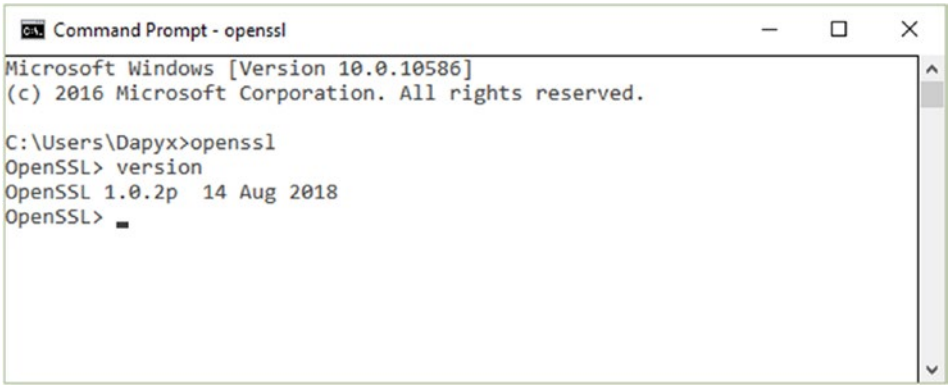


Figure 8-24. *Checking OpenSSL, second step*

Installing OpenSSL on Linux – Ubuntu Flavor

Usually, OpenSSL comes already installed on Linux – Ubuntu. For this step-by-step guide we used Ubuntu 18.04.3 LTS version (codename: bionic).

Before proceeding with the installation, check the version of OpenSSL already installed by running `openssl version -a` in the terminal. See Figure 8-25.

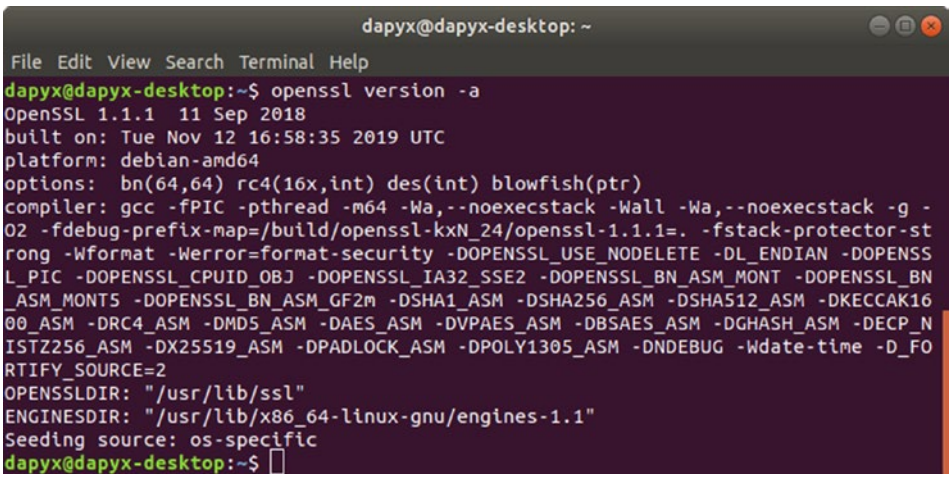


Figure 8-25. *Checking the OpenSSL version*

If you don't see it, it means that the OpenSSL was not installed or configured properly. Proceed as follows to install and configure OpenSSL.

Step 1: Update the Ubuntu system to the latest packages by running the following command in the terminal: `sudo apt-get update && sudo apt-get upgrade`. You will be asked to answer with `Y` or `N` in order to continue. Choose `Y (Yes)`. See Figure 8-26.

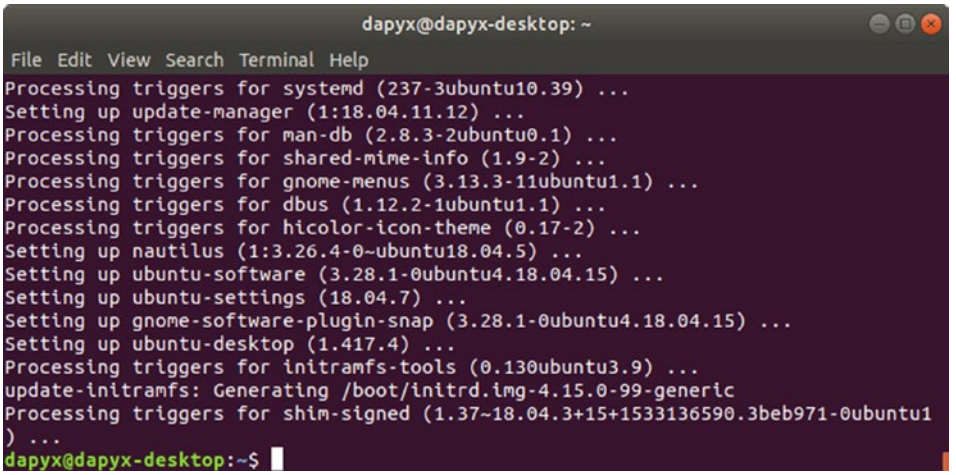


Figure 8-26. *Updating the Ubuntu system with the latest packages*

Step 3: Install the packages required for compiling. This is a very vital step. Proceed with caution. The command is `sudo apt install build-essential checkinstall zlib1g-dev -y`. See Figure 8-27.

Figure 8-27. *Installing the packages required for compilation*

Step 4: Download OpenSSL. At the moment of writing this chapter, the version was 1.1.1g. For this, follow the commands shown below and in Figure 8-28 and Figure 8-29.

```
cd /usr/local/src/
```

and

```
sudo wget https://www.openssl.org/source/openssl-1.1.1g.tar.gz
```

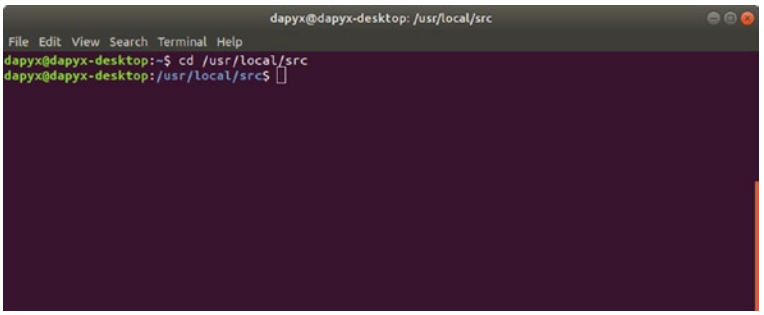


Figure 8-28. *Location of OpenSSL installation*

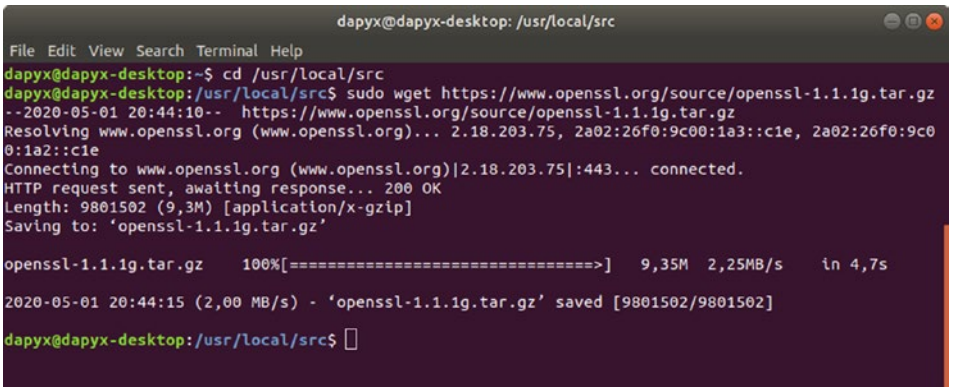


Figure 8-29. *Getting the right package for installation*

Step 5: Extract the downloaded file. To achieve this, use the following command (shown in Figure 8-30):

```
sudo tar -xf openssl-1.1.1g.tar.gz.
```

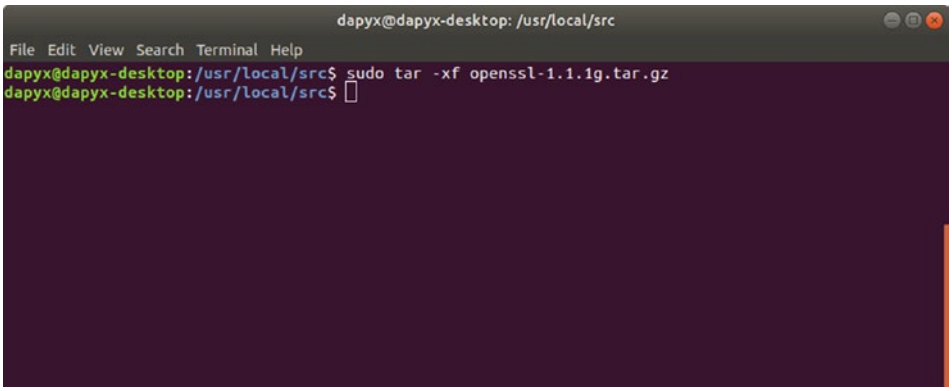


Figure 8-30. *Extracting the downloaded file*

Step 6: Navigate to the directory where the file has been extracted. See Figure 8-31.

Figure 8-31. *Location of the extracted file*

Step 7: Install OpenSSL using the following commands (shown in Figures 8-32 through 8-35):

```
sudo ./config -prefix =/usr/local/ssl -openssldir=/usr/local/ssl shared zlib
```

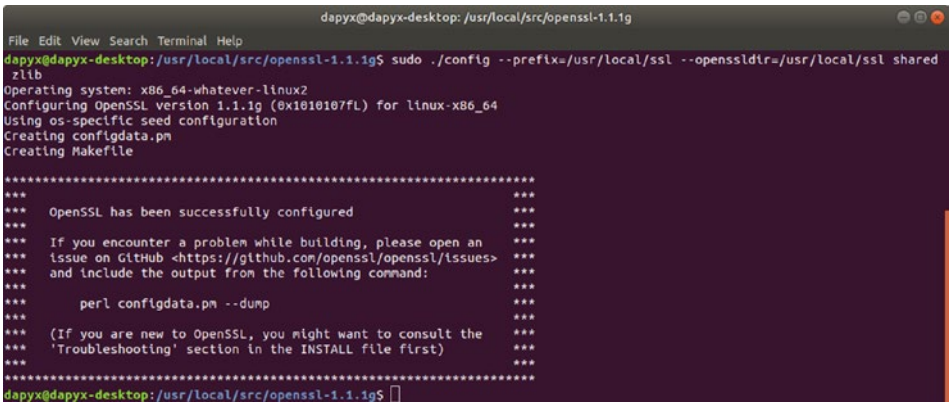


Figure 8-32. *OpenSSL installation*

```
sudo make
```

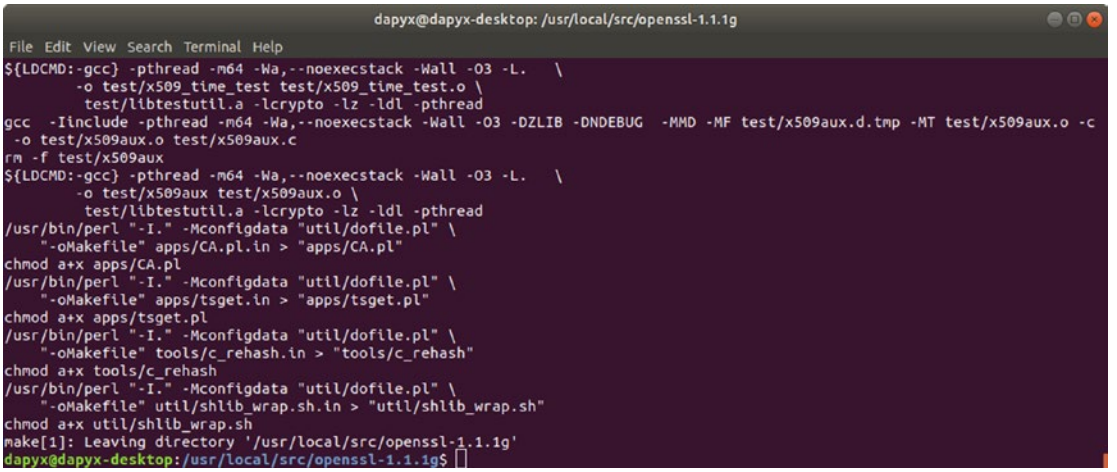


Figure 8-33. *Running sudo make*

```
sudo make test
```

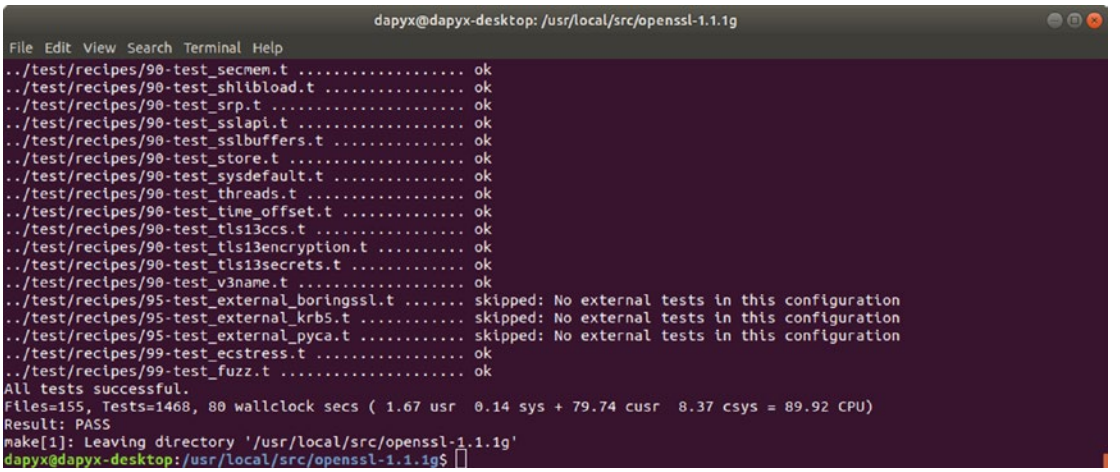


Figure 8-34. *Running sudo make test*

```
sudo make install
```

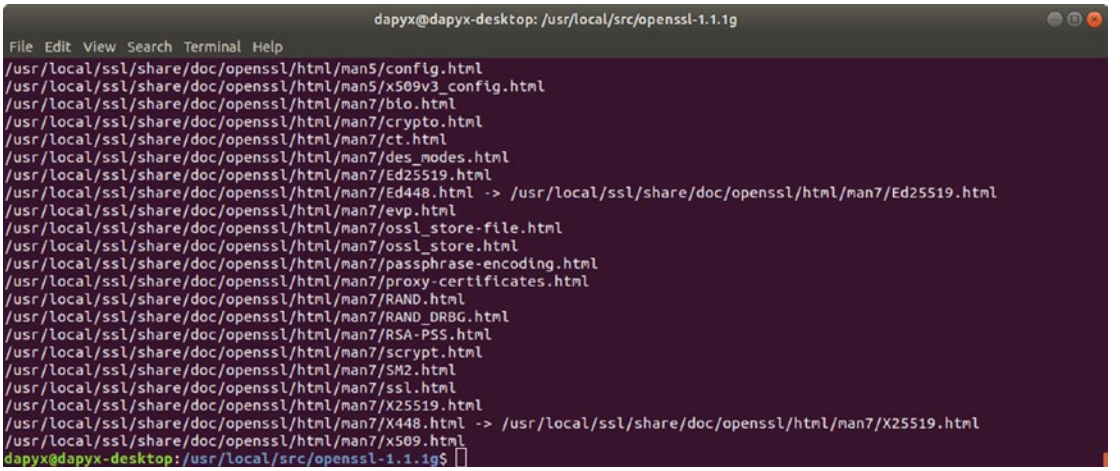


Figure 8-35. *Running sudo make install*

Step 8: Configure the OpenSSL shared libraries. First, you need to navigate to the `/etc/ld.so.conf.d` directory and create the configuration file `openssl-1.1.1g.conf` manually. The commands are as follows (shown in Figures 8-36 through 8-39):

```
cd /etc/ld/so/conf.d/
```

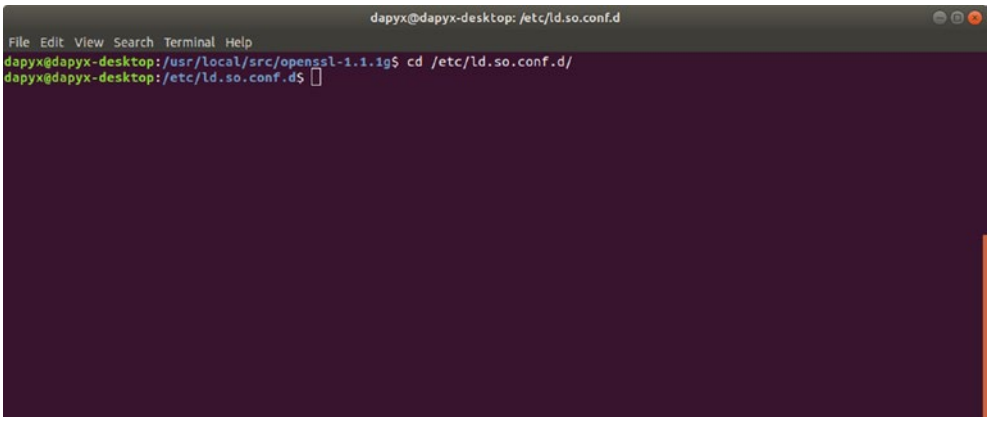


Figure 8-36. *OpenSSL shared libraries configuration*

```
sudo nano openssl-1.1.1g.conf
```

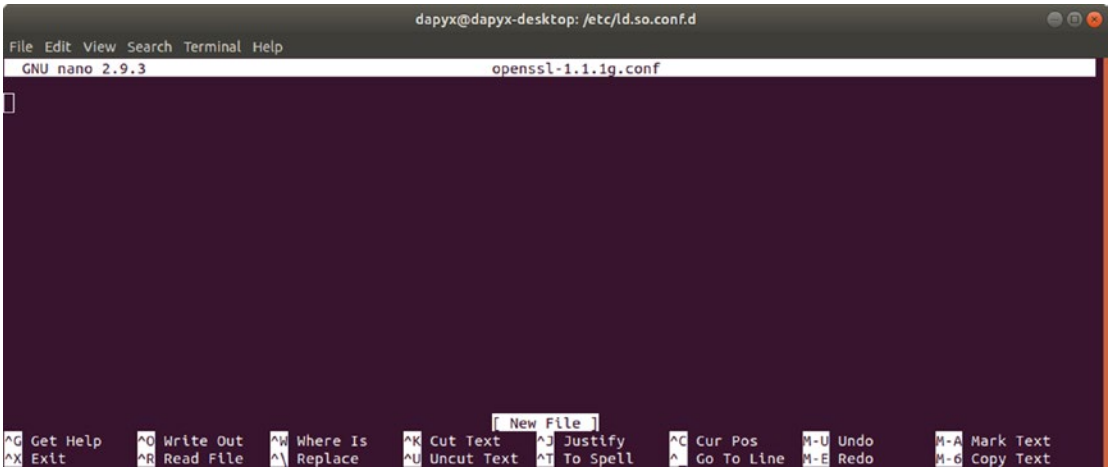


Figure 8-37. *Editing the configuration file*

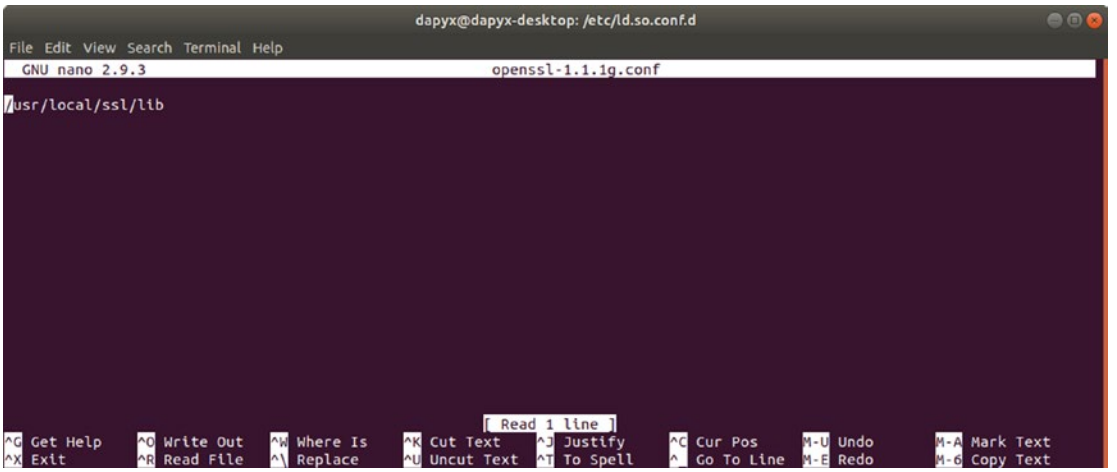


Figure 8-38. *Editing the configuration file*

```
sudo ldconfig -v
```

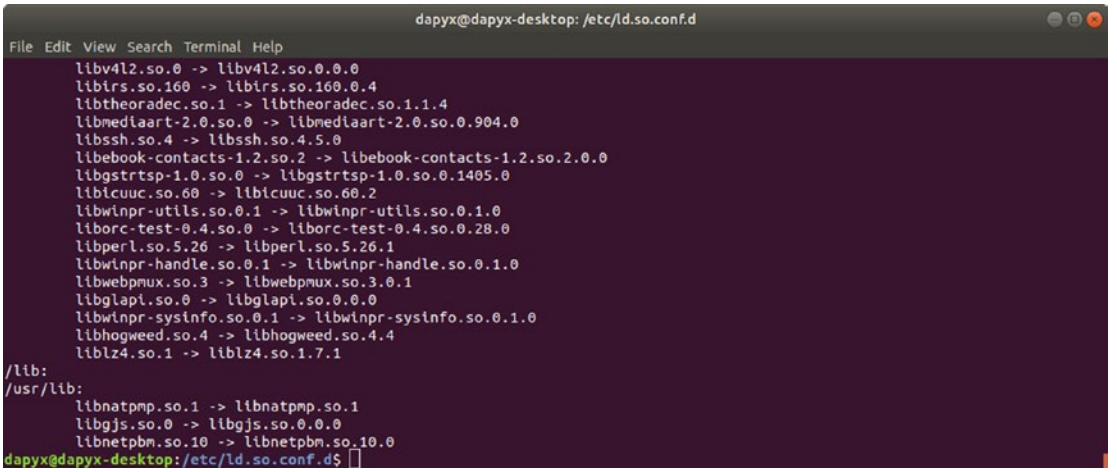


Figure 8-39. *Verifying the configuration file*

Step 9: Configure the OpenSSL binary. This step is very sensitive. You add the binary or the new version of OpenSSL installed (which is located at `/usr/local/ssl/bin/openssl`) over the default OpenSSL binary.

First, you create a backup of the binary files by running the following command:

```
sudo mv /usr/bin/c_rehash /usr/bin/c_rehash.backup
```

Second, you edit the `/etc/environment` file using vim. See Figure 8-40.

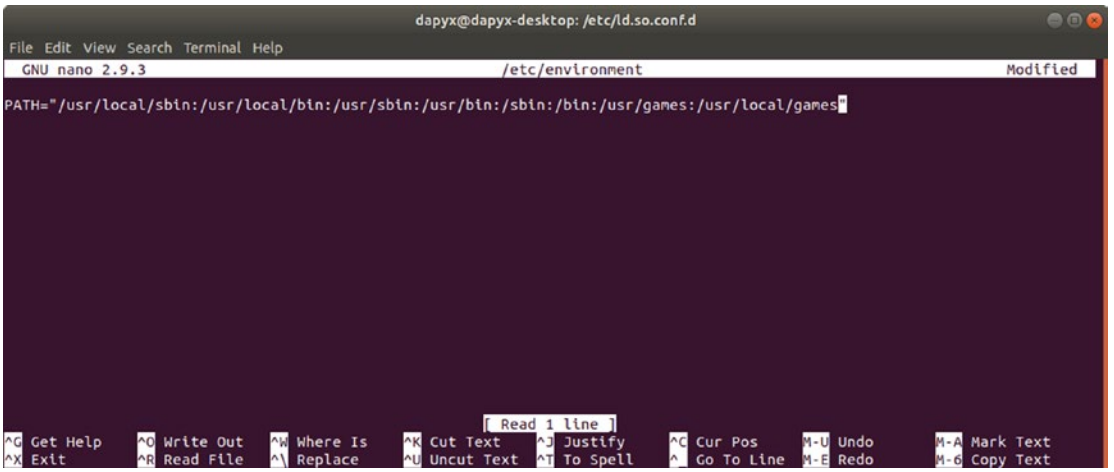


Figure 8-40. *Editing the environment path*

Ensure that you have save the file before you exit.

Third, reload the OpenSSL environment and verify the PATH `bin` directory using the following commands (shown in Figure 8-41):

```
source /etc/environment
echo $PATH
```

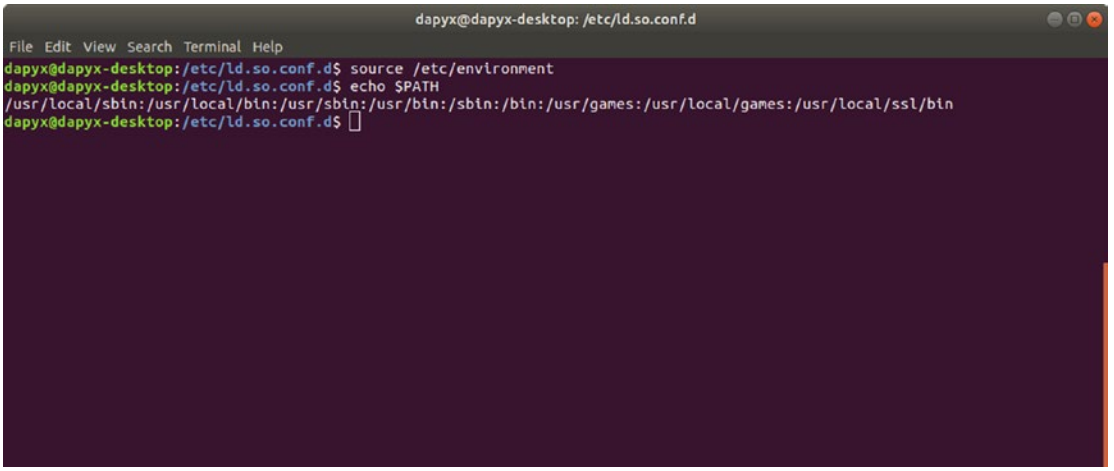


Figure 8-41. *Reloading the environment path*

For the final step, verify the installation of the last stable version of OpenSSL by using the commands from below and shown in Figure 8-42:

```
which openssl
openssl version -a
```

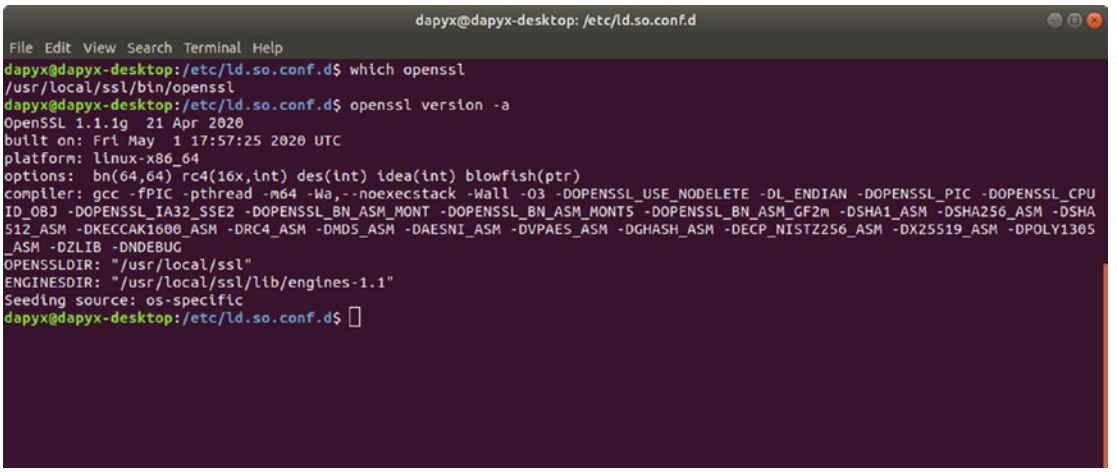


Figure 8-42. *Verifying the installation*

Botan

Botan [5] represents another powerful library that can be used in the command line as OpenSSL. The algorithms are quite vast and contain very powerful and modern implementations (including C++20 features). The feature of Botan that differentiates it from the rest of the libraries is the modules that are implemented for the Transport Layer Security (TLS) protocol. The features that are implemented with Botan made it a real candidate for inspiration and guidance among professionals. The documentation is easy to follow.

The commands and instruction are the same as the ones from OpenSSL with minor differences related to public key algorithms.

CrypTool

A great software product for cryptography developed using C++ is CrypTool (CT) [6], version 1. The latest stable release for version CT1 is 1.4.41 and it can be downloaded from CrypTool's official website[1]. After downloading, launch the executable and follow the instruction to install it. When CT1 is opened, the main window looks like Figure 8-43.

[1] www.cryptool.org/en/ct1-downloads

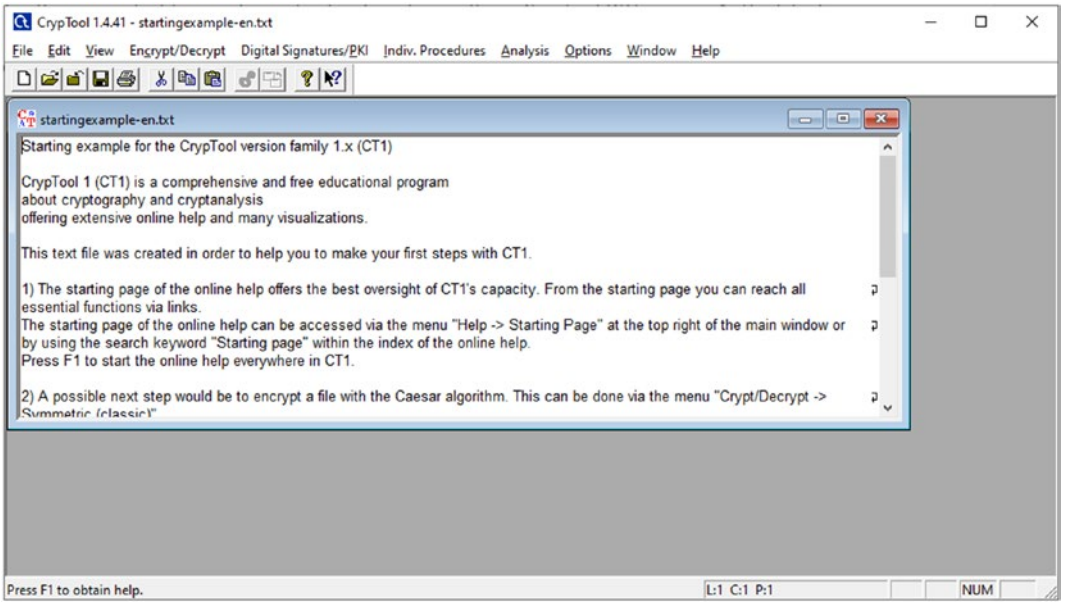


Figure 8-43. *Main window in CrypTool 1*

The first example that we will look at is the classical cipher, Caesar. It can be selected from *Encrypt/Decrypt* ➤ *Symmetric (classic)* ➤ *Caesar/Rot - 13…* Before selecting the Caesar cipher, first close the `startingexample-en.txt` window and open a new clean window from *File* ➤ *New*. In the opened window, type the sentence `This is an example of Caesar cipher using CrypTool 1.` See Figure 8-44.

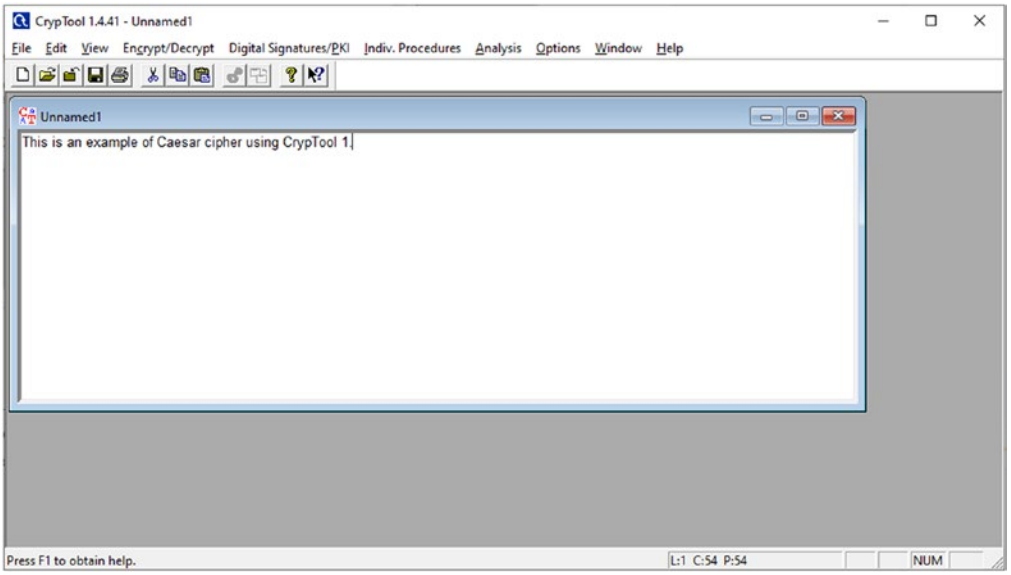


Figure 8-44. *The text in a new CT1 window*

Open the Settings window for the Caesar cipher, as described above. It should look like Figure 8-45a.

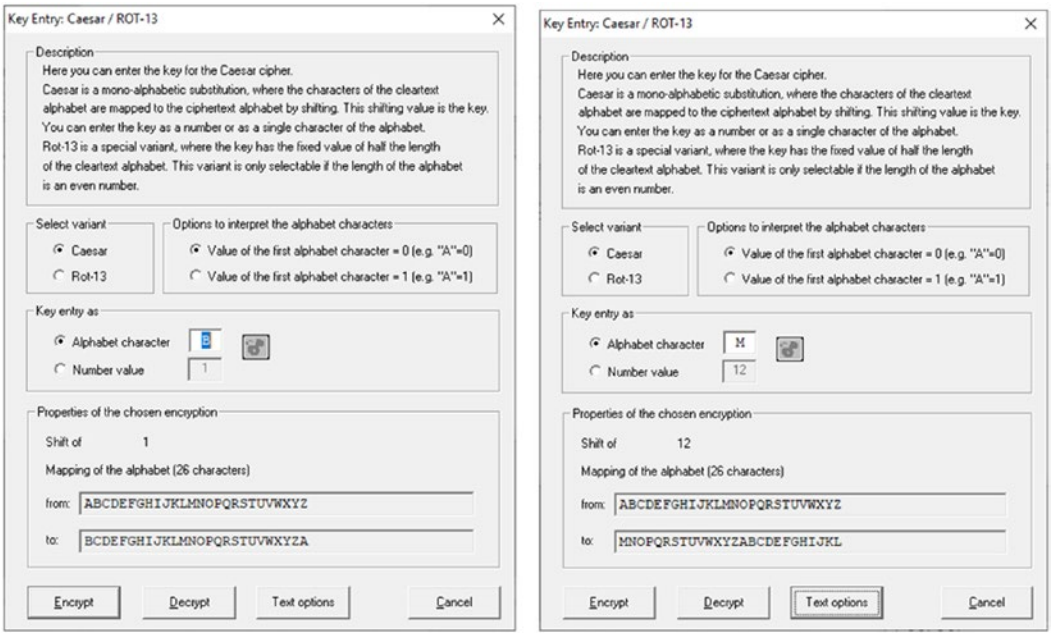


Figure 8-45. *(a) The default settings for the Caesar cipher. (b) The chosen settings for the Caesar cipher*

The window contains a short description of the cipher. Note that Rot-13 is a particular case of the Caesar cipher that shifts a particular letter 13 positions (considering that the number of the letters in the English alphabet is 26, then its half is 13, hence the name of Rot-13). Use the default variant, Caesar. On the right side, you can choose the index for the first letter of the alphabet, `A`; it can be either 0 or 1. Further, you should choose the key, which represents the number of positions a particular letter is shifted to the right in the alphabet. The key can be an alphabet letter or a number. Chose the character option and let's say the key is `M`. Figure 8-45b shows the changes in the `Properties of the chosen encryption` section. Observe that `A` is mapped to `M` (0 is the position of `A`, which is shifted 12 positions, i.e. 0+12=12; the 12th letter of the English alphabet is M) and so on. Now press the *Encrypt* button. You should obtain the result shown in Figure 8-46.

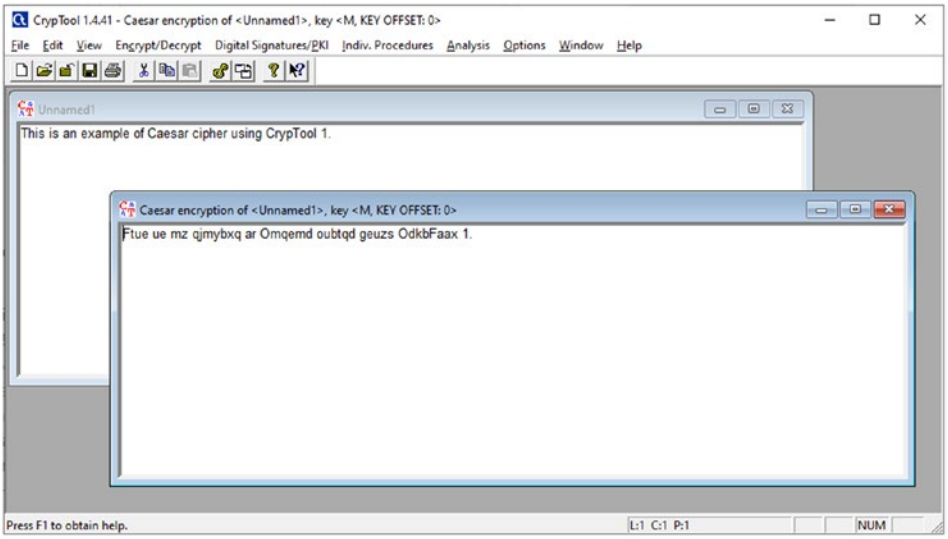


Figure 8-46. *Encryption using the Caesar cipher*

Note that the cipher is not case sensitive. Such additional settings can be accessed by pressing *Text Option* from the *Key Entry: Caesar/ROT-13* window and can be seen in the left-hand window in Figure 8-47. In this window, you keep unchanged the characters that are not in the alphabet. Note that the characters 1 and . and even the spaces were not encrypted. Further, you can choose uppercase sensitivity, you can extend the alphabet, and you can set a reference for statistical use (see the right-hand window in Figure 8-47).

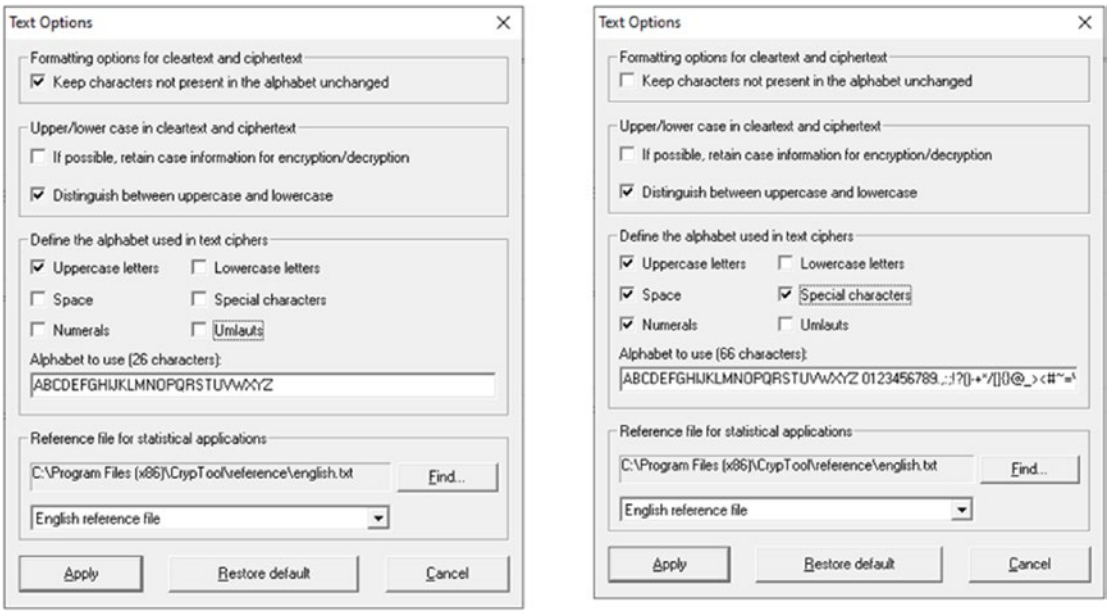


Figure 8-47. *Text options*

Now let's return to the example. Close the *Unnamed-1* window (or make sure that the emphasized window is *Caesar encryption of <Unnamed1>, key <M, KEY OFFSET: 0>* and let's decrypt the result of the Caesar encryption obtained in Figure 8-47. For this, choose the Caesar cipher from the menu, and pick the same settings as in the encryption. Note that Caesar is a symmetric cipher, which means that the same key is used for both encryption and decryption, therefore you set the key entry as an alphabet character and choose `M` and press the Decrypt button. The result is shown in Figure 8-48.

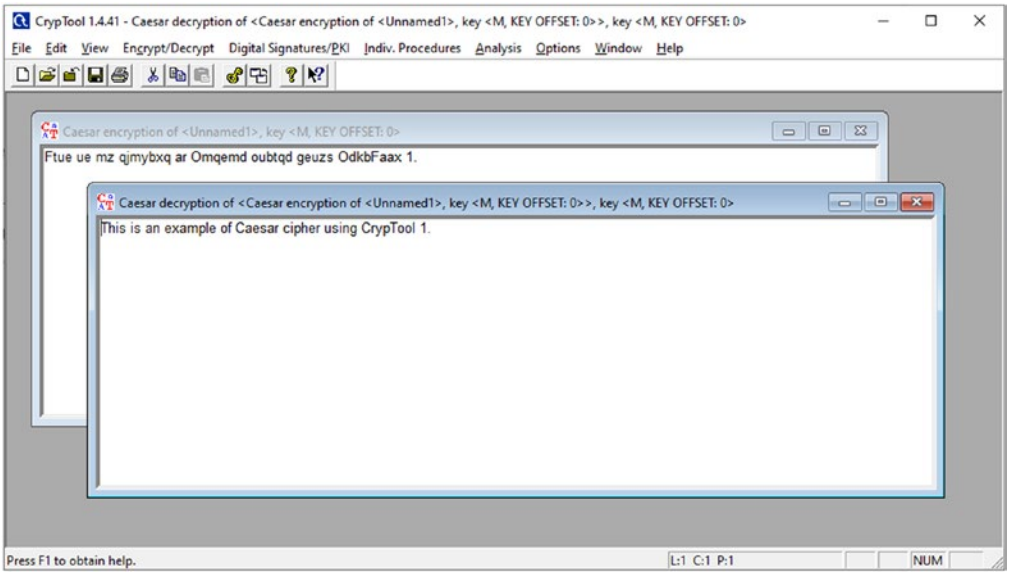


Figure 8-48. *The decryption using a Caesar cipher*

The next encryption system we will look at is RSA. Choose *Encrypt/Decrypt* ➤ *Asymmetric* ➤ *RSA Demonstrator*. The *RSA Demonstrator* window should look like Figure 8-49.

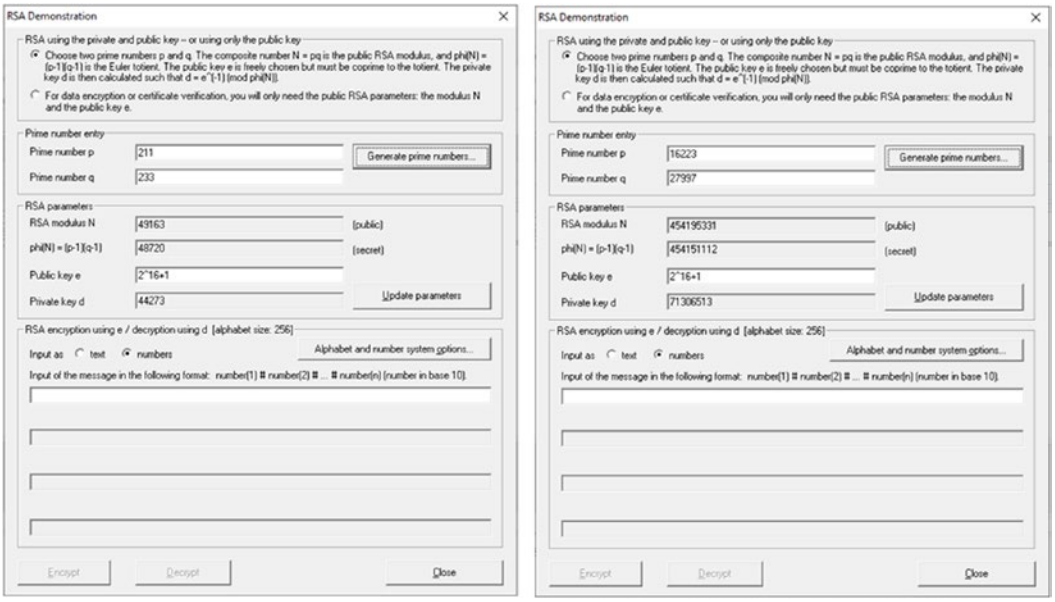


Figure 8-49. *RSA demonstration*

Keep the default option of computing both the public key and the private key. Choose the parameters for the scheme. You can provide two prime numbers yourself, or you can generate two prime numbers using the generator. Click the *Generate prime numbers* button. The window should look like the left-hand window in Figure 8-50.

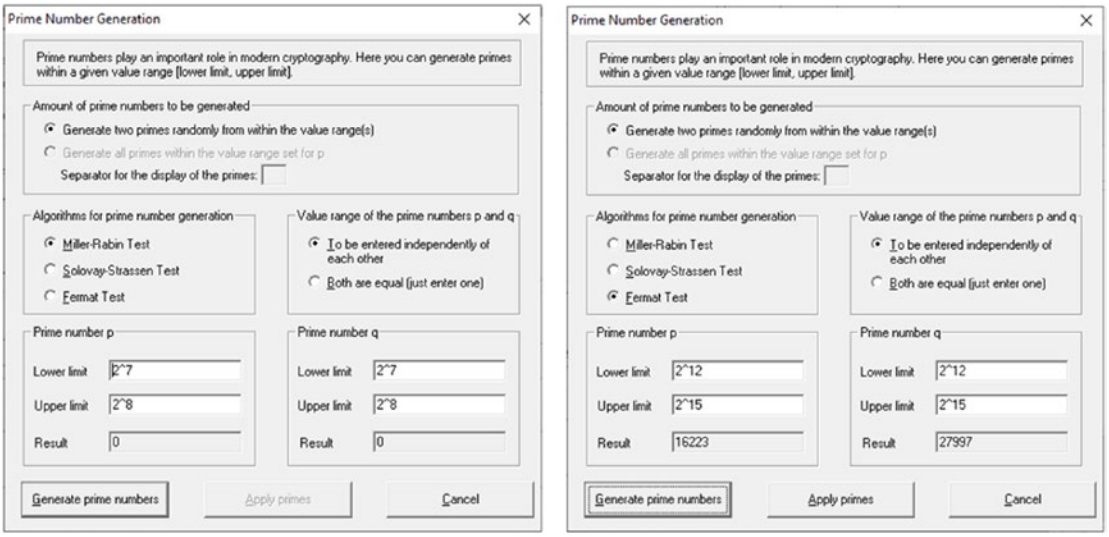


Figure 8-50. *The prime number generator for RSA*

Here, you can choose between three prime generators. Choose `Fermat Test`, set the lower limit to 2^{12} and the upper limit to 2^{15} for both `p` and `q` and opt for independent primes, and then press the *Generate prime numbers* button (right-hand window in Figure 8-50). To use these prime numbers, press the *Apply primes* button. After generating the prime number, note that the public and secret values were computed (left-hand window in Figure 8-50). Keep the default public key as $2^{16} + 1$, check the *text* option for the *Input* field, and type this: `This text is encrypted using RSA.` Press the Encrypt button. The result should look like Figure 8-51.

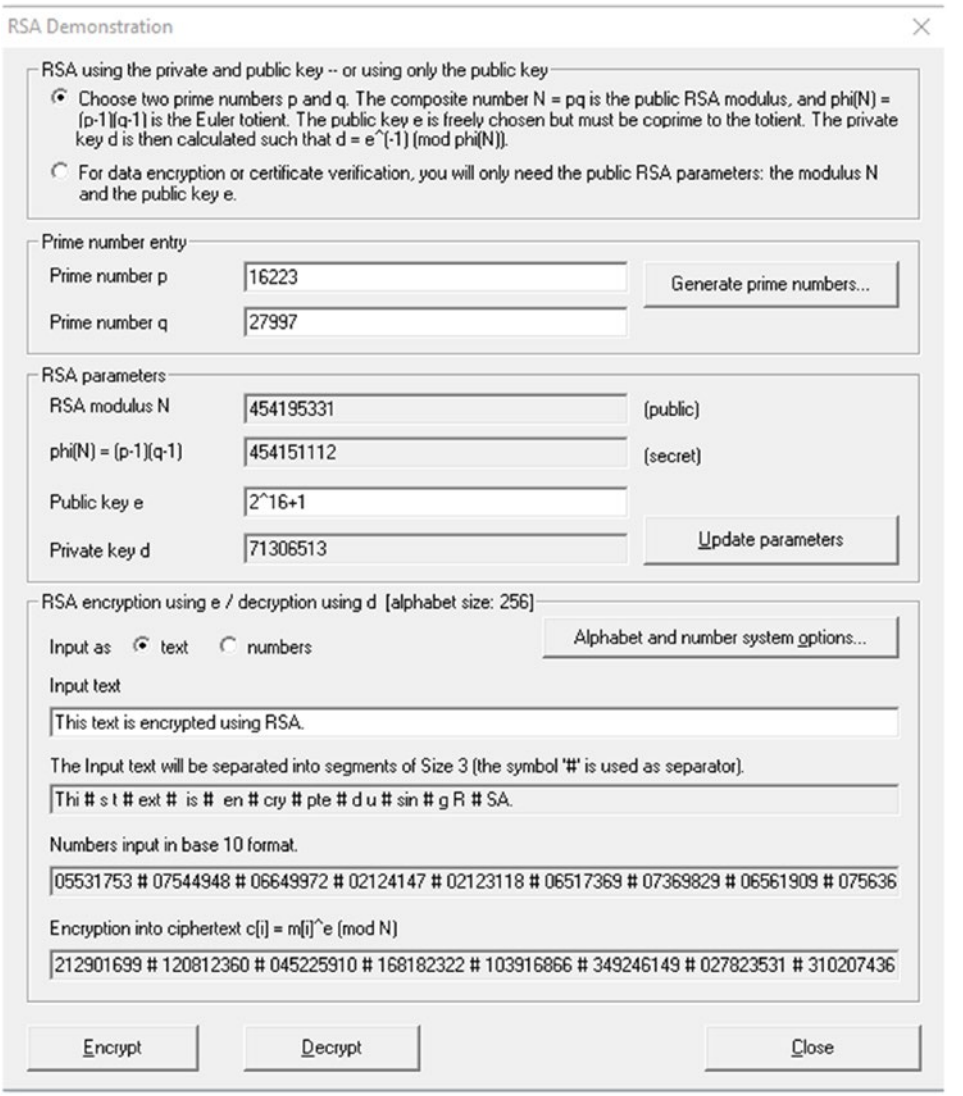


Figure 8-51. *Encrypted text using RSA*

Now, to decrypt, you should not close the window and you need to be a little careful. For decryption, copy the text resulted in the field: *Encryption into ciphertext c[i]=m[i]^e (mod N)*. Your encrypted text should look as follows (and shown in the left-hand window of Figure 8-52):

```
212901699 # 120812360 # 045225910 # 168182322 # 103916866 # 349246149 #
027823531 # 310207436 # 009232756 # 131763739 # 089946941
```

Further, check *numbers* for the *Input* and paste it. Then press the *Decrypt* button. The decryption is shown in the right-hand window of Figure 8-52. You should obtain the same plain text that you encrypted previously.

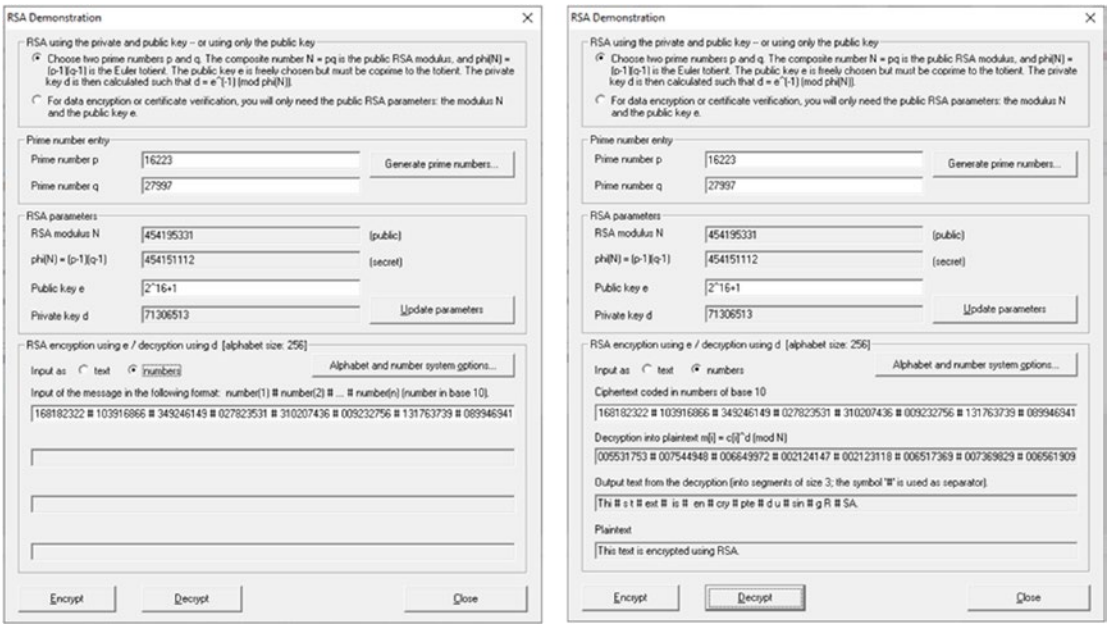


Figure 8-52. *Decryption using RSA*

These are just two simple examples of how to use CrypTool 1. It provides many more encryption schemes and examples, and it can be used for attack simulations or to collect different statistical information.

Conclusion

In this chapter, we provide a brief list of C++ libraries and we showed how to install them on the Windows operating system or on Ubuntu. The most useful libraries developed in C++ are OpenSSL, Botan, and CrypTool.

By the end of this chapter, you learned the following:

- How to access the most important open source cryptography libraries and frameworks
- How the main cryptographic operations work and how you can interact with those libraries and frameworks

- How you can access their source code with the goal of comparing the implementations of the algorithms
- How to learn from other professional developers (e.g. OpenSSL, Botan, etc.) the best practices for developing cryptographic algorithms

References

[1] OpenSSL: Cryptography and SSL/TLS Toolkit. Available online: www.openssl.org/.

[2] OpenSSL TLS/SSL and Crypto Library. Available online: https://github.com/openssl/openssl.

[3] Win32/Win 64 OpenSSL Installer for Windows. Available online: https://slproweb.com/products/Win32OpenSSL.html.

[4] OpenSSL Sources. Available online: www.openssl.org/source/.

[5] Botan: Crypto and TLS for Modern C++. Available online: https://botan.randombit.net/.

[6] CrypTool Portal. Available online: www.cryptool.org/en/.

[7] Crypto++. Available online: https://cryptopp.com/.

[8] Crypto++ Manual. Available online: https://cryptopp.com/docs/ref/.

[9] Libcrypt. Available online: www.gnupg.org/related_software/libgcrypt/.

[10] Libcrypt. Available online: www.gnupg.org/documentation/manuals.html.

[11] GnuTLS. Available online: https://gnutls.org/.

[12] GnuTLS Documentation. Available online: https://gnutls.org/documentation.html.

[13] Cryptlib. Available online: www.cryptlib.com/.

PART II

Pro Cryptography

CHAPTER 9

Elliptic-Curve Cryptography

Elliptic-curve cryptography (ECC) represents a public-key cryptography approach. It is based on the algebraic structure of elliptic curves over finite fields. ECC can be used in cryptography applications and primitives, such as *key agreement, digital signature,* and *pseudo-random generators.* It can also be used for operations such as encryption through a combination between key agreements with a symmetric encryption scheme. Some other interesting usages can be seen in several types of integer factorization algorithms that are based on elliptic curves (EC), with applications in cryptography, such as Lenstra Elliptic-Curve Factorization (L-ECC) [1]. Elliptic curves appeared for the first time in Diophantus' works [3], and it is a subject that has remained close to Diophantine geometry [2].

The starting point of elliptic-curve cryptography starts in public key cryptography (PKC). Using PKC in ECC, we have a dedicated, special case of manipulating the points of the elliptic curve and how they are generated. The manipulation consists of two cases, *multiplication* and *addition.*

The *main advantage* of ECC is that we can obtain a certain level of security based on using shorter keys, comparing with most other cryptographic algorithms that would require more resources for the same level of security.

The *second advantage* is that in some cases, elliptic-curve cryptography is quite resistant against certain attacks. These attacks are designed and developed with respect to integer factorization and discrete logarithms, and have proved to be unsuccessful.

Before proceeding further with a practical implementation, some basic theoretical notions will be presented in order to get you familiarized with elliptic-curve cryptography notions and how they work. The following section describes the required notions that will be found as well in the implementation section, where you'll find Listing 9-1 and Listing 9-2.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_9

Theoretical Fundamentals

In this section, we will describe the main foundation that must be understood before proceeding further with the practical implementation. The graphical content and the representations of some of the equations are taken and cited from [4].

Let's start with the following example, in which we assume that we have a collection of balls and we arrange them to look like a regular pyramid, in such way that on the top level we have only one ball, on the next level we have four balls, on the next level nine balls, and so on (see Figure 9-1).

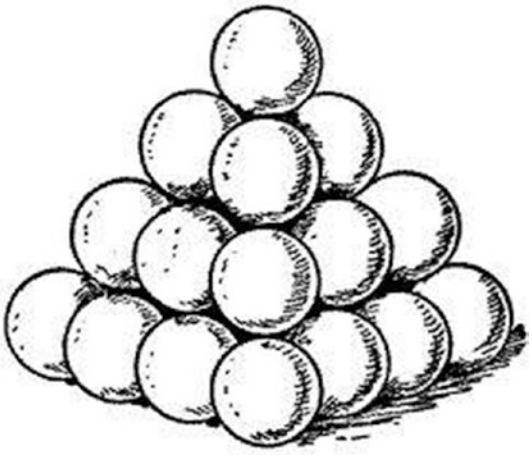


Figure 9-1. *Pyramid of balls [4]*

One logical question you might ask is, if the pyramid collapses, is there a way of rearranging the balls into a squared matrix? If the pyramid has only three levels, the rearranging process cannot be done because there are 1 + 4 + 9 = 19 *balls*, which is not a perfect square. If we have a single ball, the pyramid will be organized with one level and a squared matrix with one line and one column.

If the pyramid has the x height, then we will have

$$1^2 + 2^2 + 3^2 + \ldots x^2 = \frac{x(x+1)(2x+1)}{6} \; balls.$$

The intention is that the number is a perfect square number. To do this, we will need to resolve the following equation:

$$y^2 = \frac{x(x+1)(2x+1)}{6} \text{ in } N.$$

Such an equation represents the ***elliptic curve equation***. See Figure 9-2.

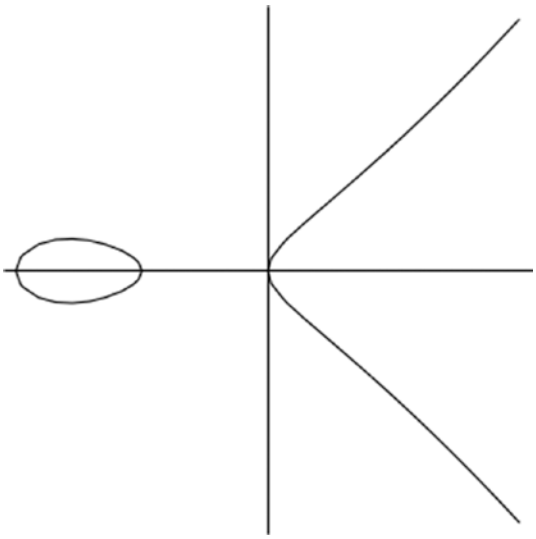


Figure 9-2. *Graphic for* $y^2 = \frac{x(x+1)(2x+1)}{6}$ *[4]*

The $y^2 = \frac{x(x+1)(2x+1)}{6}$ equation shown in Figure 9-2 can be solved using the Diophantus method, using points we know to find other points. Using $(0, 0)$ and $(1, 1)$ points, we can obtain the following straight equation: $y = x$. When we intersect the obtained curve with the equation of the line, we get the following relation:

$$x^2 = \frac{x(x+1)(2x+1)}{6} = \frac{1}{3}x^3 + \frac{1}{3}x^2 + \frac{1}{6}x$$

which is equivalent to

$$x^3 - \frac{3}{2}x^2 + \frac{1}{2}x = 0.$$

We know already two roots of this equation, $x = 0$ and $x = 1$, which are the coordinates on the *Ox* axis of the intersection points between the equation of the line and the curve. For three real numbers, *a*, *b*, *c*, we know

$$(x-a)(x-b)(x-c) = x^3 + (a+b+c)x^2 + (ab+ac+bc)x - abc.$$

In our situation, for roots 0, 1, *x* we will obtain $0+1+x = \frac{3}{2}$, finding the coordinate point $\left(\frac{1}{2}, \frac{1}{2}\right)$. Because of the symmetry of the curve, we have also the coordinate point $\left(\frac{1}{2}, -\frac{1}{2}\right)$.

Continuing with the technique illustrated above for points $\left(\frac{1}{2}, -\frac{1}{2}\right)$ and (1, 1), we will obtain the equation of the line $y = 3x - 2$, which will intersect the given curve, getting the following:

$$(3x-2)^2 = \frac{x(x+1)(2x+1)}{6}$$

equivalent to

$$x^3 - \frac{51}{2}x^2 + \ldots = 0.$$

We already know the roots $\frac{1}{2}$ and 1, so we will obtain

$$\frac{1}{2} + 1 + x = \frac{51}{2},$$

from which we have $x = 24$ and $y = 70$, which means

$$1^2 + 2^2 + 3^2 + \ldots + 24^2 = 70^2.$$

If there are 4900 balls, they can be arranged as a pyramid with a height of 24 and they can be arranged in a squared pyramid with 24 lines and 24 columns.

Weierstrass Equation

In the next section, the practical solution will be provided using the Weierstrass equation.

Definition 9-1. Let's consider elliptic curve E as being the following set: $\{(x, y)|y^2 = x^3 + Ax + B\}$, in which the elements A, B, x, y are elements from the field K, defined as $K \in \{\mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{Z}_P, \mathbb{Z}_Q\}$, where p represents a prime number, $q = p^k$, $k \geq 1$, and A, B are constants.

Definition 9-2. An equation that is defined according to *Definition 9-1* is called a ***Weierstrasss equation.***

Definition 9-3. If K is a field and $A, B \in K$, we will say that E is defined over the field K. For the points that have their coordinates in $L \subseteq K$, we will write $E(L)$. By definition, to this set we will add a point that doesn't belong to the affine plane, a point that is noted with ∞:

$$E(L) = \{\infty\} \cup \{(x, y) \in L \times L, y^3 = x^3 + Ax + B\}$$

Intuitively, it is useful to think of the graph of the elliptic curve over the field of real numbers. This has two basic forms, as shown in Figure 9-3. The equation $y^2 = x^3 - x$ has three real roots, and the equation $y^2 = x^3 + x$ has one single real root. It is not allowed to have multiple roots so we need to mention the following condition: $4A^3 + 27B^2 \neq 0$.

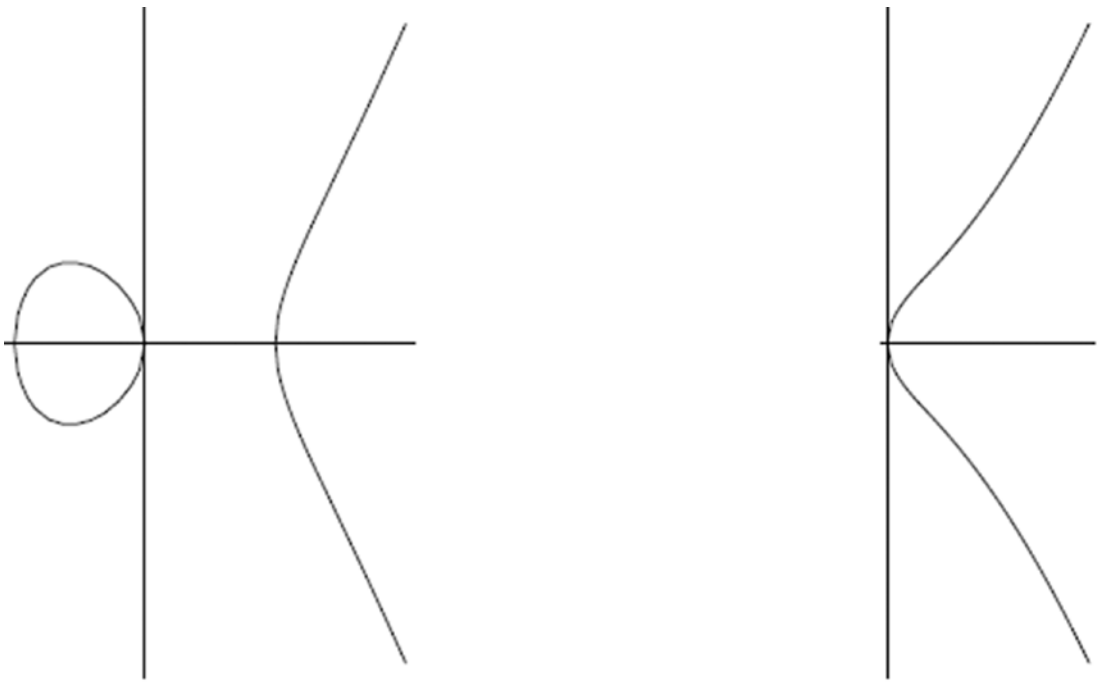


Figure 9-3. *The basic two forms of the elliptic curve over a real numbers field [4]*

If the roots are r_1, r_2, r_3, then

$$\left((r_1 - r_2)(r_1 - r_3)(r_2 - r_3)\right)^2 = -\left(4A^3 + 27B^2\right).$$

Definition 9-4. The general form of an elliptic equation over a *K* field is called the *Weierstrass generalized equation* and it has the form

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1 \ldots a_6$ are constants from *K*. This form is very useful, especially when we proceed later with the implementation.

The generalized Weierstrass equation is useful for fields with two or three characteristics. For fields with a different characteristic, we will obtain

$$\left(y + \frac{a_1x}{2} + \frac{a_3}{2}\right)^2 = x^3 + \left(a_2 + \frac{a_1^2}{4}\right)x^2 + \left(a_4 + \frac{a_1a_3}{2}\right)x + \left(\frac{a_3^2}{4} + a_6\right),$$

which is equivalent to

$$y_1^2 = x^3 + a_2'x^2 + a_4'x + a_6',$$

with $y_1 = y + \frac{a_1}{2}x + \frac{a_3}{2}$ and a_2', a_4', a_6' being constants. For fields with characteristic different than three, we have

$$x_1 = x + \frac{a_2'}{3}.$$

We will obtain

$$y_1^2 = x_1^3 + Ax_1 + B,$$

where *A* and *B* are constants.

Group Law

When it comes to a practical implementation, group law is very important for working with operations between points. There is a theorem that needs to be followed in order to have a proper implementation. Theorem 9-1 describes the properties of an elliptic curve. The properties have been implemented in Listing 9-2.

Theorem 9-1. Adding points on elliptic curve E has the following properties:

1. (Commutativity) $P_1 + P_2 = P_2 + P_1, \forall P_1, P_2 \in E$;
2. (Neutral element) $P + \infty = P, \forall P \in E$;
3. (Inverse existence) $\forall P \in E, \exists P' \in E$ in such way that $P + P' = \infty$. The P' point is noted usually with $-P$.
4. (Associativity) $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3), \forall P_1, P_2, P_3 \in E$.

Practical Implementation

This section discuss the practical implementation of ECC using C++20 and provides a basic implementation of ECC step by step.

The example (see Figure 9-4, Listing 9-1, and Listing 9-2) that we have provided represents the implementation of an elliptic curve over a finite field with order P. The following elliptic curve equation will be used for our implementation:

$$y^2 \ mod \ P = x^3 + ax + b \ mod \ P.$$

The implementation is structured in two parts:

- *Implementation of the Field Finite Element Engine* (`FFE_Engine.hpp`) – *Listing 9-1*: The file contains the signatures for the following operations and functions:
  - `int ExtendedGreatestCommongDivisor()`: The function computes the extended greater common divisor.
  - `int InverseModular()`: The function's purpose is to solve the linear congruence equation $x \times z = = 1$ (mod n);.
  - `FFE operator-() const`: The operator represents the negation operation.
  - `FFE& operator=(int i)`: The operator deals with the assignation with an integer.
  - `FFE<P>& operator=(const FFE<P>& rhs)`: The operator for assignation from the field element

- `FFE<P>& operator*=(const FFE<P>& rhs)`: Implementation for the *= operator for assignation from the field element
- `friend bool operator==(const FFE<P>& lhs, const FFE<P>& rhs)`: Implementation of the == operator for assignation from the field element
- `friend FFE<P> operator/(const FFE<P>& lhs, const FFE<P>& rhs)`: Implementation for the / operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator+(const FFE<P>& lhs, const FFE<P>& rhs)`: Implementation for the + operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator-(const FFE<P>& lhs, const FFE<P>& rhs)`: Implementation for the - operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator+(const FFE<P>& lhs, int i)`: Implementation for the a + int operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator+(int i, const FFE<P>& rhs)`: Implementation for the *int* + *a* operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator*(int n, const FFE<P>& rhs)`: Implementation for the $int * a$ operator for assignation from the field element as form `(x,y)`.
- `friend FFE<P> operator*(const FFE<P>& lhs, const FFE<P>& rhs)`: Implementation for the $a * b$ operator for assignation from the field element as form `(x,y)`.
- `template<int T>`

  `friend ostream& operator<<(ostream& os, const FFE<T>& g)`: The operator `ostream` is used for showing and displaying in readable format.

- *The main program,* listed in *Listing 9-2*: The file contains the main implementation for elliptic-curve cryptography. In the main program, a special focus is on the implementation of the operators listed above. Another important aspect for this implementation is that at the beginning of the program you can observe that the curve is defined over a finite field (a Galois field) and that any point within the elliptic curve is formed from two elements that are within the Galois fields. These points are created once there is a declaration instance of the elliptic curve itself. To perform the elliptic curve implementation, we need the following two declarations: `typedef EllipticCurve<OrderFFE_EC> this_t` and `typedef class EllipticCurve<OrderFFE_EC>::EllipticCurvePoint point_t`. Once we have these declarations, we can proceed further with the representation of the Weierstrass equation as $y^2 = x^3 + ax + b$, as represented below through the constructor of the `EllipticCurve` class:

```
//** the Weierstrass equation as y^2 = x^3 + ax + b
EllipticCurve(int CoefA, int CoefB)
     : ECParameterA(CoefA),
      ECParameterB(CoefB),
      tableOfPoints(),
      tableFilledComputated(false)
{}
```

The next step is to compute the points and to set `true` for the `tableFilledComputated` Boolean variable, used to indicate if the table with points has been filled or not for further computation. The rest of the functions are pretty straightforward and represent basic cryptographic operations between Alice and Bob, and also Oscar (the malicious third party who will try to decrypt the message).

Figure 9-4. *The output of the example*

Listing 9-1. Implementation of the Field Finite Element Engine (FFE_Engine)

```
namespace EllipticCurveCryptography
{
        //** basic functions for
        //** Finite Field Elements (FFE)
```

```
namespace HelperFunctionFFE
{
    //** Computing Extended GCD gives g = a*u + b*v
    int ExtendedGreatestCommongDivisor(int a, int b,
                                       int& u, int &v)
    {
        u = 1;
        v = 0;
        int g = a;
        int u1 = 0;
        int v1 = 1;
        int g1 = b;
        while (g1 != 0)
        {
            //** division using integers
            int q = g/g1;
            int t1 = u - q*u1;
            int t2 = v - q*v1;
            int t3 = g - q*g1;
            u = u1; v = v1; g = g1;
            u1 = t1; v1 = t2; g1 = t3;
        }

        return g;
    }

            //** providing solution and solving
            //** the linear congruence equation
            //** x * z == 1 (mod n) for z
            int InvMod(int x, int n)
            {
                  //** "%" represents the remainder
                  //** function, 0 <= x % n < |n|
                  x = x % n;
                  int u,v,g,z;
                  g = ExtendedGreatestCommongDivisor(x, n,u,v);
```

```
            if (g != 1)
            {
                   //** x and n has to be primes
                   //** in order to exist an x^-1 mod n
                   z = 0;
            }
            else
                z = u % n;

            return z;
        }
    }

     //** represents the element from a Galois field
     //** we will use a specific behaviour for the
     //** modular function in which (-n) mod m will
     //** return a negative number.
     //** The implementation is done in such way that
     //** it will offer a support for the basic
     //** arithmetic operations, such as:
     //** + (addition), - (subtraction), / (division)
     //** and scalar multiplication.
     //** The P served as an argument represents the
     //** order for the field.
     template<int P>
     class FFE
      {
      int i_;

    void assign(int i)
    {
        i_ = i;
        if ( i<0 )
        {
            //** The correction behaviour
            //** is important.
            //** Using (-i) mod p we will make sure
```

```
        //** that the behaviour is the proper one.
        i_ = (i%P) + 2*P;
    }

    i_ %= P;
}

public:
    //** the constructor
    FFE()
     : i_(0)
    {}

    //** another constructor
    explicit FFE(int i)
    {
        assign(i);
    }

    //** copying the constructor
    FFE(const FFE<P>& rhs)
     : i_(rhs.i_)
    {
    }

    //** providing access to
    //** the raw integer
    int i() const { return i_; }

    //** implementation for negation operator
    FFE  operator-() const
    {
        return FFE(-i_);
    }
```

```
//** assignation  assign from integer
FFE& operator=(int i)
{
    assign(i);
    return *this;
}

//** assignation from from field element
FFE<P>& operator=(const FFE<P>& rhs)
{
    i_ = rhs.i_;
    return *this;
}

//** implementation of "*=" operator
FFE<P>& operator*=(const FFE<P>& rhs)
{
    i_ = (i_*rhs.i_) % P;
    return *this;
}

//** implementation of "==" operator
friend bool operator==(const FFE<P>& lhs,
                       const FFE<P>& rhs)
{
    return (lhs.i_ == rhs.i_);
}

//** implementation of "==" operator
friend bool operator==(const FFE<P>& lhs,
                       int rhs)
{
    return (lhs.i_ == rhs);
}
```

```
//** implementation of "!=" operator
friend bool operator!=(const FFE<P>& lhs, int rhs)
{
    return (lhs.i_ != rhs);
}

// implementation of "a/b" operator
friend FFE<P> operator/(const FFE<P>& lhs,
                        const FFE<P>& rhs)
{
    return FFE<P>( lhs.i_ *
        HelperFunctionFFE::InvMod(rhs.i_,P));
}

   //** implementation of "a+b" operator
friend FFE<P> operator+(const FFE<P>& lhs,
                        const FFE<P>& rhs)
{
    return FFE<P>( lhs.i_ + rhs.i_);
}

//** implementation of "a-b" operator
friend FFE<P> operator-(const FFE<P>& lhs,
                        const FFE<P>& rhs)
{
    return FFE<P>(lhs.i_ - rhs.i_);
}

// implementation of "a + int" operator
friend FFE<P> operator+(const FFE<P>& lhs, int i)
{
    return FFE<P>( lhs.i_+i);
}
```

```
        //** implementation of "int + a" operator
        friend FFE<P> operator+(int i, const FFE<P>& rhs)
        {
            return FFE<P>( rhs.i_+i);
        }

        //** implementation of "int * a" operator
        friend FFE<P> operator*(int n, const FFE<P>& rhs)
        {
            return FFE<P>( n*rhs.i_);
        }

        //** implementation of "a * b"
        friend FFE<P> operator*(const FFE<P>& lhs,
                                const FFE<P>& rhs)
        {
            return FFE<P>( lhs.i_ * rhs.i_);
        }

        //** the operator ostream for
        //** showing and displaying in
        //** readable format
        template<int T>
        friend ostream& operator<<(ostream& os,
                                   const FFE<T>& g)
        {
            return os << g.i_;
        }
    };
}
```

Listing 9-2. Implementation of the Main Program

```
//** Leave everything as it is.
//** Don't change the order of the inputs or namespaces.

#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

#include <math.h>
#include "FFE_Engine.hpp"

namespace EllipticCurveCryptography
{
        //** Elliptic Curve over a finite field of order P:
        //** y^2 mod P = x^3 + ax + b mod P
        template<int OrderFFE_EC> class EllipticCurve
        {
            public:
                //** this curve is defined over the finite
                //** field (Galois field) Fp, this is the
                //** typedef of elements in it
                typedef FFE<OrderFFE_EC> ffe_element;

                 //** any point on elliptic curve is formed
                 //** from two elements that are within Fp
                 //**field (Galois Field). The points are
                 //** created once we declare an instance of
                 //** Elliptic Curve itself.
                 class EllipticCurvePoint
                 {
                     friend class EllipticCurve<OrderFFE_EC>;
                     typedef FFE<OrderFFE_EC> ffe_element;
                     ffe_element xCoordValue_;
                     ffe_element yCoordValue_;
                     EllipticCurve *ellipticCurve_;
```

```
//** core of the doubling multiplier
//** algorithm (see below)
//** multiplies acc by m as a series of
//** "2*accumulatorContainer's"
void DoublingMultiplierAlgorithm(int
            multiplier, EllipticCurvePoint&
            accumulatorContainer)
{
   if (multiplier > 0)
   {
       EllipticCurvePoint doublingValue =
                       accumulatorContainer;
       for (int counter=0; counter <
                        multiplier; ++counter)
       {
         //** doubling step
          doublingValue += doublingValue;
       }
       accumulatorContainer =
                    doublingValue;
   }
}

//** Implementation of doubling
//** multiplier algorithm.
//** The process stands on multiplying
//** intermediateResultAccumulator for
//** storing the intermediate
//** results with inputScalar.
//** This is done through
//** expansion in multipliple
//** by 2 between the first of the
//** binary represtantion of inputScalar.
```

```
EllipticCurvePoint MultiplyUsingScalar(int
      inputScalar, const EllipticCurvePoint&
      intermediateResultAccumulator)
{
    EllipticCurvePoint
           accumulatorContainer =
           intermediateResultAccumulator;
    EllipticCurvePoint outputResult =
                 EllipticCurvePoint(0,0,
                       *ellipticCurve_);
    int i = 0, j = 0;
    int iS = inputScalar;

    while(iS)
    {
        if (iS&1)
        {

 //** Setting up the bit.
 //** The computation is done by following the formula:
 //** accumulatorContainer = 2^(i-j)*accumulatorContainer
DoublingMultiplierAlgorithm(i-j,accumulatorContainer);

outputResult += accumulatorContainer;

 //** last setting for the bit
j = i;
        }
        iS >>= 1;
        ++i;
    }
    return outputResult;
}

//** the function deals with
//** adding two points on the curve
```

```
//** xCoord1, yCoord1, xCoord2=x2,
//** yCoord2=y2
void ECTwoPointsAddition(ffe_element
            xCoord1, ffe_element yCoord1,
            ffe_element xCoord2, ffe_element
            yCoord2, ffe_element & xCoordR,
            ffe_element & yCoordR) const
{
    //** dealing with sensitives cases
    //** for implying addition identity
    if (xCoord1==0 && yCoord1==0)
    {
        xCoordR = xCoord2;
        yCoordR = yCoord2;
        return;
    }
    if (xCoord2==0 && yCoord2==0)
    {
        xCoordR = xCoord1;
        yCoordR = yCoord1;
        return;
    }
    if (yCoord1==-yCoord2)
    {
        xCoordR = yCoordR = 0;
        return;
    }

    //** deal with the additions
    ffe_element s;
```

```
    if (xCoord1 == xCoord2 && yCoord1 == yCoord2)
    {
        //** computing 2*P
        s = (3*(xCoord1.i()*xCoord1.i()) +
                     ellipticCurve_->a()) /
                    (2*yCoord1);
        xCoordR = ((s*s) - 2*xCoord1);
    }
    else
    {
        //** computing P+Q
        s = (yCoord1 - yCoord2) / (xCoord1
                                   - xCoord2);
        xCoordR = ((s*s) - xCoord1 -
                                     xCoord2);
    }

    if (s!=0)
    {
        yCoordR = (-yCoord1 + s*(xCoord1 -
                                   xCoordR));
    }
    else
    {
        xCoordR = yCoordR = 0;
    }
}

EllipticCurvePoint(int xPoint, int yPoint)
: xCoordValue_(xPoint),
  yCoordValue_(yPoint),
  ellipticCurve_(0)
{}
```

```
        EllipticCurvePoint(int xPoint, int yPoint,
                    EllipticCurve<OrderFFE_EC> &
                    EllipticCurve)
         : xCoordValue_(xPoint),
           yCoordValue_(yPoint),
           ellipticCurve_(&EllipticCurve)
        {}

        EllipticCurvePoint(const ffe_element&
                    xPoint, const ffe_element& yPoint,
                    EllipticCurve<OrderFFE_EC> &
                    EllipticCurve)
         : xCoordValue_(xPoint),
           yCoordValue_(yPoint),
           ellipticCurve_(&EllipticCurve)
        {}

    public:
        static EllipticCurvePoint ONE;

        //** constructor
        EllipticCurvePoint(const
                    EllipticCurvePoint& rhsPoint)
        {
            xCoordValue_ = rhsPoint.xCoordValue_;
            yCoordValue_ = rhsPoint.yCoordValue_;
            ellipticCurve_ =
                        rhsPoint.ellipticCurve_;
        }

        //** the assignment process
        EllipticCurvePoint& operator=(const
                    EllipticCurvePoint& rhsPoint)
        {
            xCoordValue_ = rhsPoint.xCoordValue_;
            yCoordValue_ = rhsPoint.yCoordValue_;
            ellipticCurve_ =
                        rhsPoint.ellipticCurve_;
```

```
    return *this;
}

//** access x component as element of Fp
ffe_element GetX() const { return
                          xCoordValue_; }

//** access y component as element of Fp
ffe_element GetY() const { return
                          yCoordValue_; }

//** calculate the order of this point by
//** brute-force additions
unsigned int
            ComputingOrderBruteForceAddition
            (unsigned int maximum_period = ~0) const
{
    EllipticCurvePoint ecPoint = *this;
    unsigned int order = 0;
    while(ecPoint.xCoordValue_ != 0 &&
              ecPoint.yCoordValue_ != 0)
    {
        ++order;
        ecPoint += *this;
        if (order > maximum_period) break;
    }
    return order;
}

//** negation operator (-) that
//** gives the inverse of a point
EllipticCurvePoint operator-()
{
    return
             EllipticCurvePoint(xCoordValue_,
                 -yCoordValue_);
}
```

```
//** equal (==) operator
friend bool operator==(const
      EllipticCurvePoint& lhsPoint,
      const EllipticCurvePoint& rhsPoint)
{
    return (lhsPoint.ec_ == rhsPoint.ec_)
            && (lhsPoint.x_ == rhsPoint.x_) &&
            (lhsPoint.y_ == rhsPoint.y_);
}

//** different (!=) operator
friend bool operator!=(const
             EllipticCurvePoint& lhsPoint, const
             EllipticCurvePoint& rhsPoint)
{
    return (lhsPoint.ec_ != rhsPoint.ec_)
             || (lhsPoint.x_ != rhsPoint.x_) ||
             (lhsPoint.y_ != rhsPoint.y_);
}

//** Implementation of a + b operator
friend EllipticCurvePoint operator+(const
             EllipticCurvePoint& lhsPoint,
             const EllipticCurvePoint& rhsPoint)
{
    ffe_element xResult, yResult;
                  lhsPoint.ECTwoPointsAddition(
                  lhsPoint.xCoordValue_,
                  lhsPoint.yCoordValue_,
                  rhsPoint.xCoordValue_,
                  rhsPoint.yCoordValue_,
                  xResult,yResult);
    return
               EllipticCurvePoint(xResult,
                         yResult,
                        *lhsPoint.ellipticCurve_);
}
```

```
    //** multiplying with scalar * int
    friend  EllipticCurvePoint operator*(int
                    scalar, const
                      EllipticCurvePoint& rhsPoint)
    {
        return
                       EllipticCurvePoint(rhsPoint).
                            operator*=(scalar);
    }

    //** Implementation of += operator
    EllipticCurvePoint& operator+=(const
                   EllipticCurvePoint& rhsPoint)
    {
                ECTwoPointsAddition(xCoordValue_,
                yCoordValue_,rhsPoint.xCoordValue_,
                rhsPoint.yCoordValue_,xCoordValue_,
                yCoordValue_);
        return *this;
    }

    //** Implementation of *= int operator
    EllipticCurvePoint& operator*=(int scalar)
    {
        return (*this =
             MultiplyUsingScalar(scalar,*this));
    }

    //** display and print the point
    //** using ostream
    friend ostream& operator <<(ostream& os,
                    const EllipticCurvePoint& p)
    {
        return (os << "(" << p.xCoordValue_ <<
                ", " << p.yCoordValue_ << ")");
    }
};
```

```
//** performing the elliptic
//** curve implementation
typedef EllipticCurve<OrderFFE_EC> this_t;
typedef class
         EllipticCurve<OrderFFE_EC>::
              EllipticCurvePoint point_t;

//** the Weierstrass equation
//** as y^2 = x^3 + ax + b
EllipticCurve(int CoefA, int CoefB)
: ECParameterA(CoefA),
  ECParameterB(CoefB),
  tableOfPoints(),
  tableFilledComputated(false)
{
}

//** compute all the points
//** (from the group of elements) for
//** Weierstrass equation. Note the
//** fact that if we are
//** having a high order for the curve,
//** the computation process
//** will take some time
void CalculatePoints()
{
    int x_val[OrderFFE_EC];
    int y_val[OrderFFE_EC];
    for (int counter = 0; counter <
         OrderFFE_EC; ++counter)
    {
        int nsq = counter*counter;
        x_val[counter] = ((counter*nsq) +
                 ECParameterA.i() * counter +
                 ECParameterB.i()) % OrderFFE_EC;
        y_val[counter] = nsq % OrderFFE_EC;
    }
```

```
    for (int counter1 = 0; counter1 <
         OrderFFE_EC; ++counter1)
    {
        for (int counter2 = 0; counter2 <
             OrderFFE_EC; ++counter2)
        {
            if (x_val[counter1] ==
                y_val[counter2])
            {
                      tableOfPoints.push_back(Ellip
                      ticCurvePoint(counter1,
                      counter2,*this));
            }
        }
    }

    tableFilledComputated = true;
}

//** obtain the point (from the group of
//** elements) for the curve
EllipticCurvePoint operator[](int n)
{
    if ( !tableFilledComputated )
    {
        CalculatePoints();
    }

    return tableOfPoints[n];
}

//** the number og the elements
//** in the group
size_t Size() const { return
               tableOfPoints.size(); }
```

```
//** the degree of the point for
//** the elliptic curve
int Degree() const { return OrderFFE_EC; }

//** the "a" parameter, as an element of Fp
FFE<OrderFFE_EC> a() const { return
                          ECParameterA; }

//** the "b" paramter, as an element of Fp
FFE<OrderFFE_EC> b() const { return
                            ECParameterB; }

//** print and show the elliptic curve in a
//** readable format using ostream  human
//** readable form
template<int ECT>
friend ostream& operator <<(ostream& os, const
                  EllipticCurve<ECT>& EllipticCurve);

//** print and display all the elements
//** of the elliptic curve group
ostream& PrintTable(ostream &os,
                    int columns=4);

private:
    typedef std::vector<EllipticCurvePoint>
                             TableWithPoints;

    //** table with the points
    TableWithPoints tableOfPoints;

    //** first parameter of the
   //** elliptic curve equation
   FFE<OrderFFE_EC> ECParameterA;

    //** second parameter of the
    //** elliptic curve equation
     FFE<OrderFFE_EC> ECParameterB;
```

```
            //** boolean value to show if the
            //** table has been computed
            bool tableFilledComputated;
};

template<int ECT>
    typename EllipticCurve<ECT>::EllipticCurvePoint
                 EllipticCurve<ECT>::EllipticCurvePoint::
                 ONE(0,0);

template<int ECT>
ostream& operator <<(ostream& os, const
                EllipticCurve<ECT>& EllipticCurve)
{
    os << "y^2 mod " << ECT << " = (x^3" << showpos;
    if ( EllipticCurve.ECParameterA != 0 )
    {
        os << EllipticCurve.ECParameterA << "x";
    }

    if ( EllipticCurve.ECParameterB.i() != 0 )
    {
        os << EllipticCurve.ECParameterB;
    }

    os << noshowpos << ") mod " << ECT;
    return os;
}

template<int P>
ostream& EllipticCurve<P>::PrintTable(ostream &os,
                                      int columns)
{
    if (tableFilledComputated)
    {
        int col = 0;
```

```
                typename
                        EllipticCurve<P>::TableWithPoints::
                        iterator iter = tableOfPoints.begin();
                for ( ; iter!=tableOfPoints.end(); ++iter )
                {
                    os << "(" << (*iter).xCoordValue_.i() <<
                    ", " << (*iter).yCoordValue_.i() << ") ";
                    if ( ++col > columns )
                    {
                        os << "\n";
                        col = 0;
                    }
                }
            }
            else
            {
                os << "EllipticCurve, F_" << P;
            }
            return os;
        }
}

namespace utils
{
    float   frand()
    {
        static float norm = 1.0f / (float)RAND_MAX;
        return (float)rand()*norm;
    }

    int irand(int min, int max)
    {
        return min+(int)(frand()*(float)(max-min));
    }
}
```

```
using namespace EllipticCurveCryptography;
using namespace utils;

int main(int argc, char *argv[])
{
    typedef EllipticCurve<163> elliptic_curve;
    elliptic_curve myEllipticCurve(1,1);

    cout << "Basic Example of using Elliptic Curve
                Cryptography using C++20. Apress, 2020\n\n";

    //** display some informations about the
    //** elliptic curve and display some of the properties
    cout << "Equation of the elliptic curve: " <<
                                    myEllipticCurve << "\n";

    //** compute the points for the elliptic
    //** curve for equation from the above
    myEllipticCurve.CalculatePoints();

    cout << "\nList of the points (x,Y) for the curve (i.e.
                                       the group elements):\n";
    myEllipticCurve.PrintTable(cout,5);
    cout << "\n\n";

    elliptic_curve::EllipticCurvePoint P = myEllipticCurve[2];
    cout << "Randomly - Point P  = " << P << ", 2P = " <<
                                        (P+P) << "\n";

      elliptic_curve::EllipticCurvePoint Q =
                                         myEllipticCurve[3];
    cout << "Randomly - Point Q = " << Q << ", P+Q = " <<
                                               (P+Q) << "\n";

    elliptic_curve::EllipticCurvePoint R = P;
    R += Q;
    cout << "P += Q = " << R << "\n";
```

```
R = P;
R += R;
cout << "P += P = 2P = " << R << "\n";

cout << "\nEncryption of the message using
               elliptic curve principles\n\n";

//** as an example we will use Menes-Vanstone
//** scheme that is based on elliptic
//** curve for message encryption
elliptic_curve::EllipticCurvePoint G = myEllipticCurve[0];
while((G.GetY() == 0 || G.GetX() == 0) ||
            (G.ComputingOrderBruteForceAddition()<2))
{
    int n = (int)(frand()*myEllipticCurve.Size());
    G = myEllipticCurve[n];
}

cout << "G = " << G << ", order(G) is " <<
             G.ComputingOrderBruteForceAddition() << "\n";

//** Suppose that Alice wish to communicate with Bob
//** Alice and its public key
int a = irand(1,myEllipticCurve.Degree()-1);

//** generating the public key
elliptic_curve::EllipticCurvePoint Pa = a*G;
cout << "Alice - Public key (Pa) = " << a << "*" << G << "
                                    = " << Pa << endl;

//** Bob and is public key
int b = irand(1,myEllipticCurve.Degree()-1);

//** the public key
elliptic_curve::EllipticCurvePoint Pb = b*G;
cout << "Bob - Public key (Pb) = " << b << "*" << G << " =
                                      " << Pb << endl;
```

```
//** Oscar - the eavesdropper and attacker
int o = irand(1,myEllipticCurve.Degree()-1);;
elliptic_curve::EllipticCurvePoint Po = o*G;
cout << "Oscar - Public key (Po) = " << o << "*" << G << "
                                    = " << Po << endl;

cout << "\n\n";

//** Alice proceed with the encryption
//** for her message and send it to Bob.
//** To achieve this, the first step is
//** to split the message into multiple
//** parts which are encoded using Galois
//** field (Fp), which is also the domain
//** elliptic curve.
int m1 = 19;
int m2 = 72;

cout << "The clear text message send by Alice to Bob: ("
                           << m1 << ", " << m2 << ")\n";

//** proceed with encryption using the key of Bob
elliptic_curve::EllipticCurvePoint Pk = a*Pb;
elliptic_curve::ffe_element c1(m1*Pk.GetX());
elliptic_curve::ffe_element c2(m2*Pk.GetY());

//** the message that is encrypted is composed from:
//** Pa - Alice public key
//** c1,c2
cout << "The message encrypted from Alice for Bob is
                represented as {Pa,c1,c2} and its content is =
                {" << Pa << ", " << c1 << ", " << c2 <<
               "}\n\n";

//** Bob compute the decryption for the message
//** received from Alice, using her public key
//** and the session value (integer b)
Pk = b*Pa;
```

```
    elliptic_curve::ffe_element m1d  = c1/Pk.GetX();
    elliptic_curve::ffe_element m2d = c2/Pk.GetY();

    cout << "\tThe message decrypted by Bob from Alice is = ("
                          << m1d << ", " << m2d << ")" << endl;

    //** Oscar will intercept the message and
    //** and he/she will try to decrypt it
    //** using his/her key
    Pk = o*Pa;
    m1d = c1/Pk.GetX();
    m2d = c2/Pk.GetY();

    cout << "\nOscar decrypt the message from Alice = (" <<
                          m1d << ", " << m2d << ")" << endl;

    cout << endl;
}
```

Conclusion

In this chapter, we discussed elliptic-curve cryptography and how it can be implemented in practice.

Since you've reached the end of this chapter, you can now

- Understand the theoretical fundamentals for implementing elliptic-curve cryptography.
- Apply theoretical mechanisms and theorems for operations with group law in practice.
- Implement the basic operations and transpose them into practical elliptic-curve cryptography.

References

[1] Lenstra Elliptic-Curve Cryptography. Available online: https://en.wikipedia.org/wiki/Lenstra_elliptic-curve_factorization.

[2] Diophantine Geometry. Available online: https://en.wikipedia.org/wiki/Diophantine_geometry.

[3] Diophantus. Available online: https://en.wikipedia.org/wiki/Diophantus.

[4] L. Washington, *Elliptic curves: number theory and cryptography*, second edition. Chapman & Hall/CRC, Taylor & Francis Group, LLC, 2008.

CHAPTER 10

Lattice-Based Cryptography

In this chapter, you will get an overview of lattice-based cryptography. You will learn why lattices are important in the cryptography field and the challenges in using them. Further, you will explore a practical implementation that uses lattices, namely the GGH (Goldreich–Goldwasser–Halevi) encryption scheme [1].

Lattices are important in cryptography because the hardness assumption based on them is considered to be quantum resistant. In the last few years, the number of primitives in quantum cryptography has increased. While the traditional encryption systems, such as RSA, Diffie-Hellman, and elliptic-curve encryption systems, can be easily broken using quantum computers, encryption systems that use lattices are one of the few candidates that can resist in post-quantum cryptography.

However, using lattices in cryptography is not an easy task regarding the applicability and the practical implementations, because they are complex mathematical constructions that require a quite solid background of algebra and an understanding of abstract concepts.

Mathematical Background

This section provides a short overview of the main elements and techniques that are required as minimum theoretical information about lattices and the mathematical background that a professional should know.

Take into consideration the space $\mathbb{R}^n$ and a base in $\mathbb{R}^n$ of the form $b = (b_1, \ldots, b_n)$, with $b_1, \ldots, b_n \in \mathbb{R}$. A lattice has the following form:

$$\mathcal{L}(b) = \left\{ \sum a_i b_i \mid a_i \in \mathbb{Z} \right\}$$

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_10

In the above construction, a_i is an integer number and b_i is the ith element of the basis b. Moreover, it can be observed that $\mathcal{L}$ is the set of all linear combinations that have integer coefficients. An immediate example of a lattice is $\mathbb{Z}^n$, generated by the standard basis in $\mathbb{R}^n$. Figure 10-1 shows a lattice in a Euclidean plane.

Figure 10-1. *A lattice in a Euclidean plane*[1]

Examples of lattice problems are: shortest vector problem (SVP), closest vector problem (CVP), shortest independent vector problem (SIVP), GapSVP, GapCVP, bounded distance decoding, covering radius problem, and shortest basis problem. In cryptography, SVP and CVP are mainly used as hardness assumptions in cryptosystems.

For SVP, the following elements are given: a vector space V, a basis b in the vector space, and a norm N. Knowing the lattice $\mathcal{L}(b)$, it is required to compute the shortest vector $v \in V$ such that v's norm in V represents the minimum distance defined in $\mathcal{L}$. In other words, the vector $v \in V$ should be found such that

$$\|v\| = \lambda(\mathcal{L}(b))$$

In the above relation, $\|.\|$ represents the norm in V, $\mathcal{L}(b)$ is the lattice defined over the basis b, and λ is the minimum distance defined in $\mathcal{L}(b)$. The relation gives the search variant of the SVP. The other two variants are

- **Calculation**: Find the minimum distance in lattice $\lambda(\mathcal{L}(b))$ when given the basis b and lattice $\lambda(\mathcal{L}(b))$.
- **Decision**: Decide whether $\lambda(\mathcal{L}(b)) \leq d$ or $\lambda(\mathcal{L}(b)) > d$ when given the basis b, lattice $\lambda(\mathcal{L}(b))$, and a real value $d > 0$.

[1]Source: https://en.wikipedia.org/wiki/Lattice_(group

A generalization of SVP is CVP, where, informally speaking, given a vector $v \in V$, it is required to find the vector u in $\mathcal{L}(b)$ which is nearest to v. Note that v is not necessarily in $\mathcal{L}(b)$. In some cases, there is an additional condition: the distance between v and u should not exceed a given value.

For more information about the lattices used in cryptography, you can consult [2], [3].

Example

In this section, we present a GGH encryption scheme [1] that uses lattices. GGH is an asymmetric encryption scheme, namely it uses the public key for encryption and the private key for decryption. The algorithms of the cryptosystem are the well-known key generation, encryption and decryption. In the following, we present them as proposed in [1]:

- **Key generation**: Given a security parameter, generate a basis b in the lattice $\mathcal{L}$ defined over an n-dimensional space that has good properties (such as containing nearly orthogonal vectors) and a unimodular matrix A. The basis and the matrix compose the private key. The public key is computed as $B = A \cdot b$.
- **Encryption**: Given the message $m = (m_1, \ldots, m_n)$ and the error $e = (e_1, \ldots, e_n)$, the encryption is $c = m \cdot B + e$.
- **Decryption**: Given the encryption $c = (c_1, \ldots, c_n)$, the message is computed in two steps:
  1. Compute $c \cdot b^{-1}$. This yields $c \cdot b^{-1} = (m \cdot B + e)b^{-1} = m \cdot A \cdot b \cdot b^{-1} + e \cdot b^{-1} = m \cdot A + e \cdot b^{-1}$.
  2. Remove $e \cdot b^{-1}$ using a technique such as Babai rounding and compute $m = m \cdot A \cdot A^{-1}$.

Further, we provide the implementation of the encryption and decryption for GGH (Listing 10-1), using as keys the following values:

$$b = \begin{pmatrix} 17 & 0 \\ 0 & 19 \end{pmatrix}; A = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}$$

The result is shown in Figure 10-2.

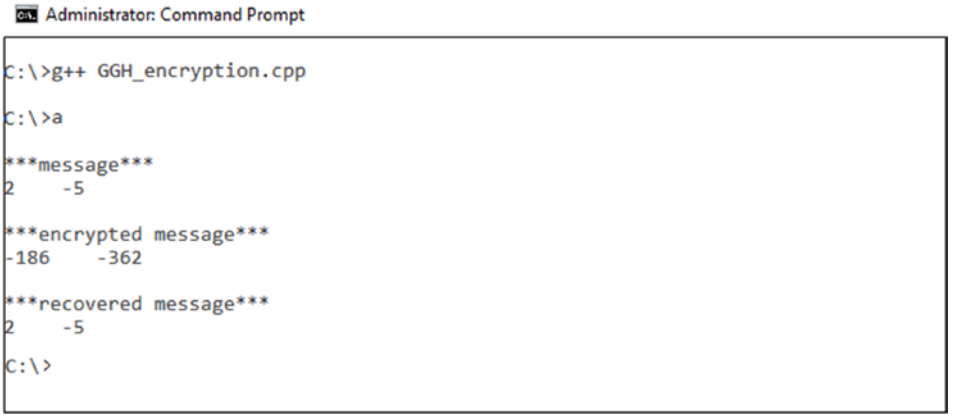


Figure 10-2. *The result of Listing 10-1*

Listing 10-1. Encryption and Decryption Algorithm of the GGH Cryptosystem

```
#include <iostream>
#include "math.h"

using namespace std;

void encrypt(double message[100], double public_B[100][100], double error_
vals[100], int dimension, double output_encrypted_text[100]);
void decrypt(int dimension, double encrypted_message[100], double private_
basis[100][100], double unimodular_matrix[100][100], double output_
message[100]);
double matrix_determinant(double square_matrix[100][100], int dimension);
void matrix_inverse(double matrix[100][100], int dimension, double output_
inverse[100][100]);
void matrix_multiplication(double matrix1[100][100], double matrix2[100]
[100], double output[100][100], int dimension) ;
void matrix_addition(double matrix1[100][100], double matrix2[100][100],
double output_sum[100][100], int dimension);
void get_cofactor(double matrix[100][100], double aux[100][100], int p,
int q, int n);
void adjoint_matrix(double matrix[100][100], double adjoint[100][100],
int dimension);
bool inverse_matrix(double matrix[100][100], double inv_matrix[100][100],
int dimension);
```

```
void vector_to_matrix(double v[100], int dimension, double output_
matrix[100][100]);
void matrix_to_vector(double matrix[100][100], double output_v[100], int
dimension);
void print_matrix(double matrix[100][100], int n, string message);
void print_vector(double vect[100], int n, string message);
void print_message(string message);

int main()
{
    int message_length = 2;

    double b[100][100] = {{17.0, 0.0}, {0.0, 19.0}}; // the private
    basis -> b
    double b_inverse[100][100];

    inverse_matrix(b, b_inverse, message_length);

    double A[100][100] = {{2.0, 3.0}, {3.0, 5.0}}; // the private
    unimodular matrix -> A
    double A_inverse[100][100];
    inverse_matrix(A, A_inverse, message_length);

    double B[100][100]; // the public key -> B
    matrix_multiplication(A, b, B, message_length);

    // Encryption
    double enc_message[100]; // stores the encryption of the message -> c
    double message[100] = {2, -5}; // the message -> m
    double error_vals[100] = {1, -1}; // the error values -> e

    print_vector(message, message_length, "message");
    encrypt(message, B, error_vals, message_length, enc_message);
    print_vector(enc_message, message_length, "encrypted message");

    // Decryption
    double recovered_message[100];
    decrypt(message_length, enc_message, b, A, recovered_message);
    print_vector(recovered_message, message_length, "recovered message");
}
```

```
// Auxiliary function that prints a matrix on the console
void print_matrix(double matrix[100][100], int n, string message)
{
    cout<<endl<<"***"<<message<<"***"<<endl;

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++ )
            cout<<matrix[i][j]<<"    ";

        cout<<endl;
    }

    cout<<endl;
}

// Auxiliary function that prints a vector on the console
void print_vector(double vect[100], int n, string message)
{
    cout<<endl<<"***"<<message<<"***"<<endl;

    for(int i = 0; i < n; i++)
    {
         cout<<vect[i]<<"    ";
    }

    cout<<endl;
}

// Auxiliary function that prints a string message on the console
void print_message(string message)
{
    cout<<endl<<"***"<<message<<"***"<<endl;
}

void encrypt(double message[100], double public_B[100][100], double error_
vals[100], int dimension, double output_encrypted_text[100])
{
    // c=m•B+e
```

```
    double aux_message[100][100], aux_enc_message[100][100], aux_error_
    vals[100][100];
    vector_to_matrix(message, dimension, aux_message);

    // Compute m•B -> aux_enc_message
    matrix_multiplication(aux_message, public_B, aux_enc_message, dimension);
    vector_to_matrix(error_vals, dimension, aux_error_vals);

    // Compute m•B+e -> output_encrypted_text
    matrix_addition(aux_enc_message, aux_error_vals, aux_enc_message,
    dimension);
    matrix_to_vector(aux_enc_message, output_encrypted_text, dimension);
}

void decrypt(int dimension, double encrypted_message[100], double private_
basis[100][100], double unimodular_matrix[100][100], double output_
message[100])
{
    // (1) Compute c * (b^(-1))
    // (2) Remove e * (b^(-1))
    // (3) Compute m * A * (A^(-1))
    double aux_enc_message[100][100], aux_message[100][100];
    double recovered_message[100][100];

    // Compute the inverse of the basis -> b_inverse
    double b_inverse[100][100];
    inverse_matrix(private_basis, b_inverse, dimension);

    // Compute the inverse of the unimodular matrix -> A_inverse
    double A_inverse[100][100];
    inverse_matrix(unimodular_matrix, A_inverse, dimension);

    // (1) Compute c * (b^(-1)) -> aux_enc_message
    vector_to_matrix(encrypted_message, dimension, aux_enc_message);
    matrix_multiplication(aux_enc_message, b_inverse, aux_message,
    dimension);
```

```
    // (2) Remove e * (b^(-1)) from aux_enc_message
    // Basically, the value aux_message[i][j] is rounded to the neareast
       integer
    for (int i=0; i<2; i++)
    {
        for (int j=0; j<2; j++)
            aux_message[i][j] = round(aux_message[i][j]);
    }

    // (3) Compute m * A * (A^(-1))
    matrix_multiplication(aux_message, A_inverse, recovered_message,
    dimension);
    matrix_to_vector(recovered_message, output_message, dimension);
}

// Computes the matrix multiplication between two matrices
void matrix_multiplication(double matrix1[100][100], double matrix2[100]
[100], double output[100][100], int dimension)
{
    for (int i = 0; i < dimension; i++)
    {
        for (int j = 0; j < dimension; j++)
        {
            output[i][j] = 0;
            for (int k = 0; k < dimension; k++)
                output[i][j] += matrix1[i][k] *  matrix2[k][j];
        }
    }
}

// Computes the matrix sum between two matrices
void matrix_addition(double matrix1[100][100], double matrix2[100][100],
double output_sum[100][100], int dimension)
{
    for(int i = 0; i < dimension; ++i)
        for(int j = 0; j < dimension; ++j)
            output_sum[i][j] = matrix1[i][j] + matrix2[i][j];
```

```
}

// Computes the cofactor of the element matrix[p][q]
void get_cofactor(double matrix[100][100], double aux[100][100], int p, int
q, int n)
{
    int i = 0, j = 0;

    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            if (row != p && col != q)
            {
                aux[i][j++] = matrix[row][col];

                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
}

// computes the determinant of a square matrix
double matrix_determinant(double square_matrix[100][100], int dimension)
{
   double matrix_det = 0.0;
   double aux_matrix[100][100];

   if (dimension == 1)
      return square_matrix[0][0];

   if (dimension == 2)
      return ((square_matrix[0][0] * square_matrix[1][1]) - (square_
      matrix[1][0] * square_matrix[0][1]));
```

```
   else
   {
      for (int k = 0; k < dimension; k++) {
         int aux_i = 0;
         for (int i = 1; i < dimension; i++) {
            int aux_j = 0;
            for (int j = 0; j < dimension; j++) {
               if (j == k)
               continue;
               aux_matrix[aux_i][aux_j] = square_matrix[i][j];
               aux_j++;
            }
            aux_i++;
         }
         matrix_det = matrix_det + (pow(-1.0, k) * square_matrix[0][k] *
         matrix_determinant( aux_matrix, dimension - 1 ));
      }
   }
   return matrix_det;
}

// Computes the adjoint of a matrix
void adjoint_matrix(double matrix[100][100], double adjoint[100][100],
int dimension)
{
    if (dimension == 1)
    {
        adjoint[0][0] = 1;
        return;
    }

    int sign = 1;
    double aux[100][100];
```

```
    for (int i=0; i<dimension; i++)
    {
        for (int j=0; j<dimension; j++)
        {
            get_cofactor(matrix, aux, i, j, dimension);
            sign = ((i + j) % 2 == 0) ? 1 : -1;
            adjoint[j][i] = (sign)*(matrix_determinant(aux, dimension - 1));
        }
    }
}

// Computes the inverse of a matrix
bool inverse_matrix(double matrix[100][100], double inv_matrix[100][100],
int dimension)
{

    double det = matrix_determinant(matrix, dimension);
    if (det == 0)
    {
        return false;
    }

    double adj[100][100];
    adjoint_matrix(matrix, adj, dimension);

    for (int i=0; i<dimension; i++)
        for (int j=0; j<dimension; j++)
        {
            if(adj[i][j] / det == -0)
                adj[i][j] = 0.0;

            inv_matrix[i][j] = adj[i][j] / det;
        }

    return true;
}
```

```
// This function "converts" a vector (seen as a matrix with 1 line and
   *dimension* columns) into a matrix
// The obtained matrix has on the first line the elements of the vector
// The remaning lines (*dimension* - 1) contanis 0
// This "conversion" is useful in the operations with matrices (addition,
   multiplication)
void vector_to_matrix(double v[100], int dimension, double output_
matrix[100][100])
{
    for(int i = 0; i < dimension; i++)
    {
        output_matrix[0][i] = v[i];
    }

    for(int i = 1; i < dimension; i++)
        for (int j = 0; j < dimension; j++)
        {
            output_matrix[i][j] = 0;
        }
}

// This function "converts" a matrix into a vector
// All lines of the matrix has values of 0, except for the first line
// The first line of the matrix becomes the vector
void matrix_to_vector(double matrix[100][100], double output_v[100], int
dimension)
{
    for(int i = 0; i < dimension; i++)
    {
        output_v[i] = matrix[0][i];
    }
}
```

Conclusion

In this chapter, we discussed lattice-based cryptography and its importance. At the end of this chapter, you now know the following:

- The importance of lattice-based cryptography and its impact on the future of cryptography
- How to encrypt and decrypt using GGH cryptosystem
- How to implement practical functions and methods related to lattices and matrices

References

[1] O. Goldreich, S. Goldwasser, and S. Halevi, "Public-key cryptosystems from lattice reduction problems" in *Annual International Cryptology Conference* (pp. 112-131). Springer; Berlin, Heidelberg. August, 1997.

[2] D. Micciancio and O. Regev, "Lattice-based cryptography" in *Post-quantum cryptography* (pp. 147-191). Springer; Berlin, Heidelberg. 2009.

[3] H. Knospe, *A Course in Cryptography* (Vol. 40). American Mathematical Society. 2019.

CHAPTER 11

Searchable Encryption

Searchable encryption (SE) is an encryption technique that allows outsourcing the encrypted data to possible untrustworthy third-party service providers, while at the same time allowing the users to apply searching operations directly over the encrypted data safely and securely. Searchable encryption can be considered a type of fully homomorphic encryption, which will be discussed in Chapter 12.

To understand the searchable encryption technique, consider the following scenario. There is a set of documents owned by *data owner* A, which is stored on a server, but these documents are allowed to be accessed (in a specific way that will be detailed immediately) by *data user* B. To keep them secure, A encrypts the documents using B's public key and then stores them on the server. In this scenario, B has permission only to search in the documents (note that the documents are in an encrypted format) or to read them (note that B can read a document only after it is retrieved from the server and decrypted). Let's say B wants to retrieve from the server any documents that contain a specific keyword, for example, "programming." To do this, B constructs a value called a *trapdoor* based on the query word "programming" and the secret key that B owns and then submits the trapdoor value to the server. The server will perform the search algorithm given by the searchable encryption scheme and will send the result (in encrypted format) to B.

Another more practical example is the following: a software solution that needs at some point the social security numbers (SSNs) of their customers is developed by an entity. The rules and good practices suggest that the SSNs be encrypted when working with them. This can be challenging because the employees will work with the SSNs, such as when they need to search for a user account. A solution is that the employees search for a particular SSN through the encrypted SSNs (without decrypting them in any way). A searchable encryption scheme would make this possible.

Now that you have a view of searchable encryption, it is worth saying that it has great potential, letting the data user search for specific content over encrypted data. An immediate application of searchable encryption is in the healthcare domain, where patients'

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_11

medical files can be searched in an encrypted form. Other applications are in education, business, and basically in any domain that requires search processes through data.

Components

The components of a searchable encryption scheme are as follows: the entities and the algorithms. In this section, we present these components in a detailed view.

Entities

When a software solution is implemented, more aspects should be clarified before the implementation itself: the clients who will use the application, the entity that will maintain it, the type of data, the roles supported by the application, and so on.

In a system that uses a searchable encryption scheme, the following entities are involved in the whole process:

- **Data owner**: The data owner, who is assumed to be a trusted party, has a number of n documents $=\{D_1, \ldots, D_n\}$, which are characterized by keywords (note that these keywords are not metadata). Both the documents and the keywords will be outsourced. Prior to outsourcing the documents (and the keywords, which are often organized into a structure called *index structure*) on the server, they are encrypted by the data owner using an encryption algorithm of a searchable encryption scheme.
- **Data user**: The data user, who is an authorized user of the data, may trigger the search process. Using the query keyword for which the search will be made, the data user generates a trapdoor value that will be used when searching over the encrypted data. Also, the data user may decrypt the documents from the search process if the data user possesses the private key. Note that the data owner can be a data user.
- **Server**: The server, which is considered semi-trusted or honest-but-curious, stores the encrypted data and performs the search algorithms based on the trapdoor value that it receives from the data user. *Semi-trusted* or *honest-but-curious* means that it performs the search algorithms as instructed but can analyze the data that was given to it.

Types

From the cryptographic point of view, searchable encryption schemes can be categorized as follows: *symmetric searchable encryption* (SSE) schemes and *public encryption with keywords search* (PEKS) schemes. Symmetric searchable encryption schemes use just a key for the encryption or decryption of the content and additionally in other specific algorithms, as you will see below. Public encryption schemes with keyword search use two keys, namely a public key to encrypt content and a private (or secret) key to decrypt the encrypted content.

The SSE schemes contain the following algorithms [1]:

- `KeyGeneration`: The data owner runs this algorithm. The input is a security parameter λ, and the output is the secret key *SK*.
- `BuildIndex`: The data owner runs this algorithm. Its purpose is to generate a structure of indices that will contain the keywords that describe the documents. The input is the secret key *SK* and the set of documents *D* that will be stored on the server, while the output is an index structure *I*. Specifically, this algorithm begins with an empty index structure and for every document of the set, it appends to the index structure some keywords that describe the current document. Note that the keywords are encrypted using the secret key *SK* in a specific way that can be different from the encryption of the documents before being added to the index structure. The index structure can be a tree, a hash table, a list, etc.
- `Trapdoor`: The data user runs this algorithm. The input for the trapdoor algorithm is the desired query keyword *kw*, for which the search process is triggered, and the secret key *SK*, while the output is a value T_{kw} called trapdoor. Note that the trapdoor algorithm does not just encrypt the query keyword *kw*. Instead, it adds a noise value or works with something of control.
- `Search`: The server runs the search algorithm. The input for the search algorithm is the trapdoor value T_{kw} obtained from the previous algorithm and the index structure *I* obtained from the `BuildIndex` algorithm. Note that the search algorithm does not just try to match the trapdoor T_{kw} in *I*. The search algorithm should specify how the trapdoor value T_{kw} is searched in the index structure (remember that T_{kw} is not a simple encryption of a plain keyword).

If the search algorithm returns one or more documents that contain the query keyword, the documents are sent to the data user, or else the server sends a proper message. Note that the encryption and decryption algorithms are not listed above. That is because two different encryption schemes can be chosen by the data owner, namely one to encrypt the documents and one for the searchable encryption scheme. This situation is possible because the searchable encryption scheme does not directly involve the documents. All algorithms of an SSE scheme work only with the keywords and/or the index structure of encrypted keywords.

A little different from the SSE version, the algorithms of a PEKS scheme are as follows [2]:

- `KeyGeneration`: This algorithm is similar to the `KeyGeneration` from SSE, and the data owner runs this algorithm, too. The input is also a security parameter λ, while this time the output of the key generation is a pair of keys, namely the public and the private keys, (PK, SK).
- `Encryption`: The data owner runs this algorithm, for which the public key PK and a keyword KW are the input values, while the output is the encrypted value SW of KW.
- `Trapdoor`: Similar to the trapdoor algorithm from SSE, the data user runs this algorithm to generate the trapdoor value. The input is the secret key SK and the query keyword KW for which the search is made, while the output is the trapdoor value T_{KW} corresponding to the keyword KW.
- `Test`: The server runs the test algorithm, for which the input is the public key PK, an encrypted value C (representing the encryption of a keyword KW'), and the trapdoor value T_{KW}. The output of the test algorithm is 1 if $KW' = KW$, and 0 otherwise.

The same remarks apply for the trapdoor and the test algorithm; the algorithms do not just offer encryption or simple matches, respectively. Still, the above algorithms for the SSE schemes and PEKS schemes are presented according to their introductions in this field in early works [1] and [2]. Since then, the algorithms were adapted alongside the options that the search process supported. Namely, multiple keywords search is allowed in some works, others enable fuzzy search (which allows small typos or inconsistencies of the format) based on keywords [3, 4], and yet others enable a semantic search

(the search process returns documents that contain keywords from the semantic field of the query keyword) [5], etc. Other works focus on the documents (specifically, the documents can be updated directly on the server, without it being necessary to retrieve them from the server, decrypt, update, encrypt, and store them again on the server); other works focus on the index structure that can be updated directly on the server [6]. However, the algorithms that are contained by any searchable encryption scheme are the trapdoor and the search/test algorithm, and, of course, the encryption and decryption.

Security Characteristics

There are some things that need to be protected in a searchable encryption such as the *search pattern* and the *access pattern*. The *search pattern* is the information that can be discovered from the fact that two different search results belong to the same query keywords. The *access pattern* is the set of documents resulting from a trapdoor corresponding to a given keyword *KW*. Besides, searchable encryption schemes should meet security requirements related to search queries, too. According to [7], SE schemes should have the following characteristics: controlled searching (search queries may be submitted only by authorized users), encrypted queries (the query search itself should be encrypted before being submitted to the server), and query isolation (the server learns nothing from the queries that it receives).

The SSE schemes should be IND1-CKA and/or IND2-CKA (chosen keyword attack for indexes) resistant, which means that the index structure cannot be compromised. In IND1-CKA, the same number of keywords is chosen for all documents used in the build index structure, while in IND2-CKA the documents can be described by a different number of keywords. On the other hand, the PEKS scheme should be resistant to the chosen keyword attack (which is a challenge between an attacker and the structure that manages the PEKS scheme).

Recently introduced security requirements are forward and backward privacy for the dynamic SE schemes, which allow inserting, updating, or deleting to be applied over the set of documents or the keywords directly on the server, without the need of decrypting them. Backward and forward privacies refer to the information that can be discovered in the process of inserting/deleting/updating. *Backward privacy* refers to the information that is discovered when the search is made for a keyword for which documents have been deleted before the current search, while *forward privacy* means that the current update operation is not related to previous operations.

An Example

The following example [18] shows that searchable encryption (SE) is a very powerful encryption technique. The advantage is that the user may search for keywords within encrypted documents. Recall that the participants to the system are the *data user*, who owns a set of documents $S = \{D_1, \ldots, D_n\}$, prepares the system by generating the keys, encrypts the documents and the keywords, and stores them on a cloud server; the *data owner*, who is allowed to submit search queries on the cloud server; and the *cloud server*, which stores the documents in an encrypted format and runs the search algorithm.

The work [18] uses elliptic curves (see Chapter 9) in the searchable encryption scheme. Nowadays, elliptic curves are used in important areas such as blockchains ([14], [15]) or the Internet of Things ([16], [17]).

Figure 11-1 [18] shows an example of a searchable encryption scheme that uses elliptic-curve cryptography and is designed for a big data environment (see Chapter 15). In the work [18], the Elliptic Curve Digital Signature Algorithm (ECDSA) is used to secure the content of courses available for students on an e-learning platform. The security parameter (λ) for the key generation algorithm of the searchable encryption scheme is the private key from the ECDSA algorithm.

At this moment, there is no practical implementation of a searchable encryption scheme that can be used in a real environment, due to the technique's complexity, although there are attempts. After an in-depth study of the current research, we could not find at this moment a practical implementation in the form of a library, module, or framework. For implementing a searchable encryption scheme, several basic guidelines should be taken into consideration before beginning the implementation:

- The architecture of the software application (server, database, services, etc.)
- The hardware components and the way they are managed for the current applications, which include security and cryptographic techniques
- The architecture should be designed such that processes within the searchable encryption are represented as independent algorithms such that their deployment is made correctly between the end users and the existing network infrastructure.

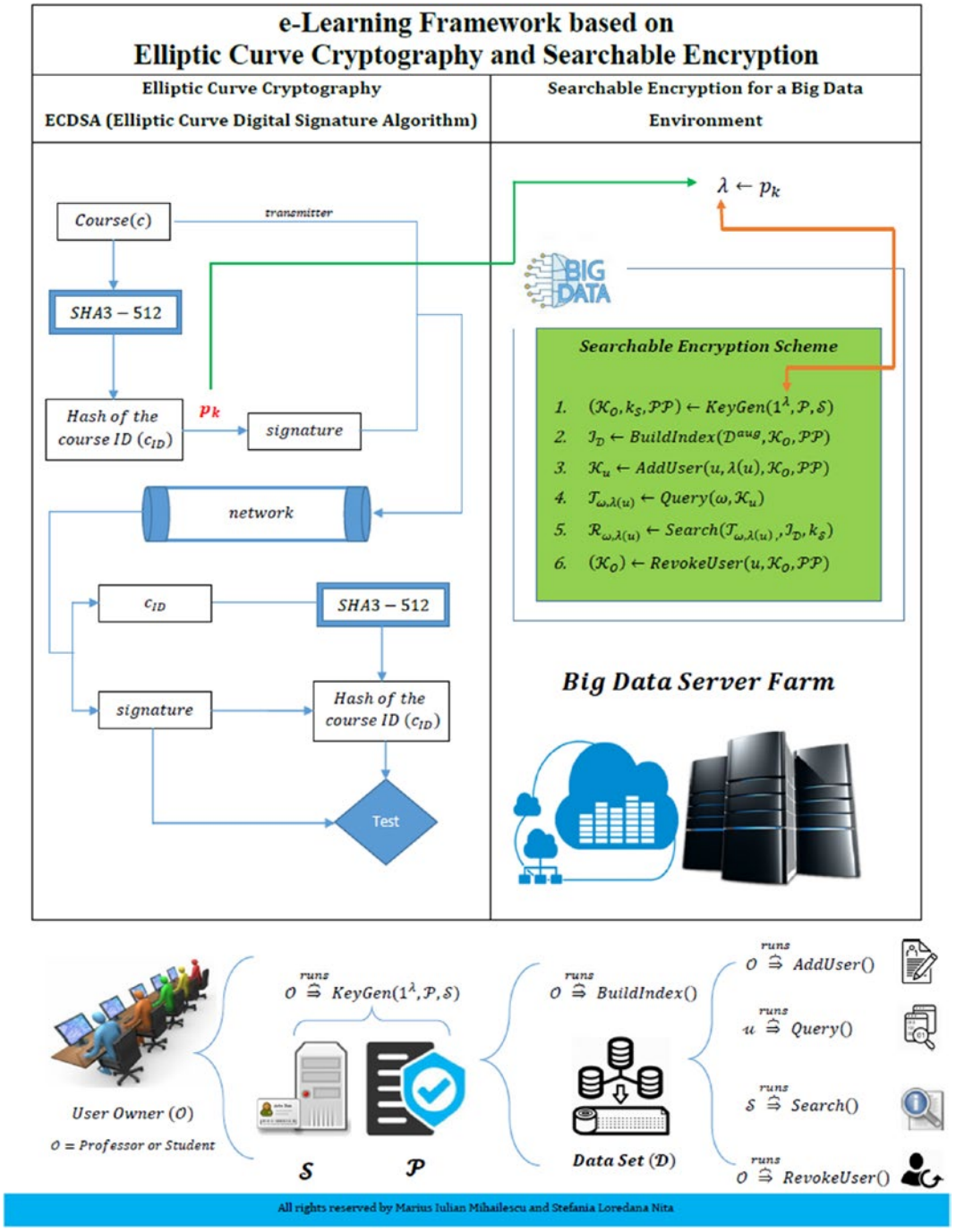


Figure 11-1. *[18]. An example of practical searchable encryption scheme*

Note that the searchable encryption scheme presented in Figure 11-1 is partitioned in more steps. Every step is an algorithm that can be considered a separate instance from the searchable encryption scheme. Further, the instances can be implemented as software modules or services or IoT devices (for example, devices like Intel NUC PC or a Raspberry PI). The distribution and deployment of the software modules or services among the users can be realized through a distributed network, for example, on a cloud computing network or a regular network for small and medium business architectures.

The algorithms below [18] show the steps from Figure 11-1, which present a searchable encryption for a big data environment. Before implementing the steps, it is necessary to understand how the steps are organized as independent algorithms. The following are the steps:

1. $(\boldsymbol{K_O, K_s, PP}) \leftarrow \boldsymbol{KeyGeneration}(\mathbf{1}^{\lambda}, \boldsymbol{P}, \boldsymbol{S})$. The data owner O runs this probabilistic algorithm for which the input values are the security parameter λ, a policy P. The output is a tuple composed of the owner's secret key K_O, the server key K_s, and the public parameters PP.

2. $\boldsymbol{I_D} \leftarrow \boldsymbol{BuildIndex}(\boldsymbol{D^{aug}}, \boldsymbol{K_O}, \boldsymbol{PP})$. The data owner O runs this probabilistic algorithm for which the input values are the description of the data set D^{aug} (namely, the keywords that describe each document) and the secret key of the owner (K_O), and the output is an index structure I_D.

3. $\boldsymbol{K_U} \leftarrow (\boldsymbol{u}, \boldsymbol{\lambda(u)}, \boldsymbol{K_O}, \boldsymbol{PP})$. The data owner O runs this probabilistic algorithm to enroll a new user in the e-learning platform system. The input values for the algorithm are the identity of the new user, the level of access of the user (user's role), and the owner's O key. The output is the secret for the new user.

4. $\boldsymbol{Trapdoor}_{(\omega, \lambda(u))} \leftarrow \boldsymbol{Query}(\boldsymbol{\omega}, \boldsymbol{K_u})$. The data user who has the proper clearance $\lambda(u)$ for generating a search query runs this probabilistic algorithm. The input values are the keyword $\omega \in \Delta$ (where Δ is a dictionary of keywords) and the user's secret key. The output is the query token (trapdoor value) $Trapdoor_{(\omega, \lambda(u))}$.

5. $\boldsymbol{R}_{(\omega, \lambda(u))} \leftarrow \boldsymbol{Searching}(\boldsymbol{Trapdoor}_{(\omega, \lambda(u))}, \boldsymbol{I_D}, \boldsymbol{K_s})$. The server ($S$) runs this probabilistic algorithm that searches the index for the data items that contain the query keyword ω. The input values are

the search query and the index, and the output is $\boldsymbol{R}_{(\omega, \lambda(u))}$, which includes a set of identifiers of the data items $d_j \in D_{\omega, \lambda(u)}$ that contain the query keyword ω such that $\lambda(d_j) \leq \lambda(u)$, where $\lambda(u_i)$ is the access level of the user that triggered the search query, or a failure symbol φ.

6. $(\boldsymbol{K_O}) \leftarrow \boldsymbol{RevokeUser}(\boldsymbol{u}, \boldsymbol{K_O}, \boldsymbol{PP})$. The data owner O runs this probabilistic algorithm to revoke a specific user from the system. The input values are the user's id, the data owner's secret keys and the server, while the output is new keys for the owner and server.

The searchable encryption scheme designed for this chapter is correct if for all $k \in N$, for all K_O, K_S outputted by $KeyGen(1^{\lambda}, P)$, for all D^{aug}, for all I_D that is outputted by $BuildIndex(D^{aug}, K_O)$, for all $\omega \in \Delta$, for all $u \in U$ for all K_u outputted by $AddUser(K_O, u, \lambda(u), PP)$, $Search(I_D, T_{\omega, \lambda(u)}) = D_{\omega, \lambda(u)}$.

Pseudocode 11-1 presents a sketch for the practical implementation of the searchable encryption scheme proposed in Figure 11-1. Note that the implementation is purely demonstrative as the implementations (frameworks, libraries, etc.) for searchable encryption do not exist at this moment.

***Pseudocode 11-1**. Guidelines for Implementing a Searchable Encryption Scheme*

```
#include <iostream>
#include <fstream>

class KeyGeneration
{
// Step 1
// The data owner runs the algorithm
// from KeyGeneration step (algorithm)

// global variables
public: string securityParameter;
        string ownerID;
        string policyContent;
        string serverIdentity;
```

```
// the function will return the policy,
// as a content or file
public: string GetPolicy(ifstream& policyContent)
{
string content = "";
if (policyContent.is_open())
{
    while (getline (policyContent, line))
    {
      content += line;
    }
    policyContent.close();
  }

  else policyContent = "Cannot read the policy file";
  return policyContent
}
// getting server identity can be tricky and it has
// different meanings, such as the name of computer,
// IP, active directory reference name etc...
// For the current example we will use the hardware ID
public: string GetServerIdentity()
{
string serverIdentity = "";

// here goes the implementation for getting the server identity
// for this method, Windows WMI can be used
// this link provides more details:
// https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-
   page?redirectedfrom=MSDN

return serverIdentity
}

// class constructor
public: KeyGeneration(){}
```

```
// let's generate the secret key, server key
// and public parameters
// "#" represents the separator
public: string ReturnParameters(KeyGeneration kp)
{
string sbParameters = "";
sbParameters += kp.ownerSecretKey + "#" + kp.serverKey + "#" +
kp.publicParameters;
return sbParameters;
}
}

class BuildIndex
{
// Step 2
// the algorithm from BuildIndex step (algorithm)
// are runned and invoked by the data owner

// constructor of the class
public: void BuildIndex(){}

// the function centralize the build index parameters
// after their initialization and processing
public: void UseBuildIndexParameters()
{
list<string> descriptionDataSet;
string ownerPrivateKey = "";
string outputIndex = "";
}

//simulation of getting the data set and their
//descriptions
public: list<string> GetDataSet()
{
list<string> ll;
```

```
for(int i = 0; i < dataSet.size(); i++)
{
ll.push_back(description[i]);
}
}

// getting the private of the owner
public: string ownerPrivateKey()
{
string privateKey = "";

// get the private key and work with it arround

return privateKey;
}

// get the index
public: string Index()
{
string index = "";

// implement the query for getting
// or generating the index

return index;
}
}

class AddUser
{
// Step 3
// the algorithm from AddUser step (algorithm)
// are runned and invoked by the data owner
// constructor of the class AddUser
public: AddUser() {}
```

```
// property for getting the identity of the user
// see below the Class Student
public: string IdentityOfTheUser()
{
string identity = "";

// implement the way of getting
// the identity of the user

return identity;
}

// property for getting the owners key
public: string OwnerSecretKey()
{
string secretKey = "";

// implement the way of querying
// for secret key

return secretKey;
}

public: void AssignSecretKeyToUser()
{
AddUser u = new AddUser();
Student stud = new Student(u.OwnerSecretKey);
}
}

class Query
{
// Step 4
// the algorithm from Query step (algorithm)
// are runned and invoked by the user

// constructor of the class Query
public: Query() {}
```

```
// function for getting the keywords
public: string Keyword()
{
string kw = "";

// query for the keywords;

return kw;
}

// function for getting the secret key of the users
public: string UserSecretKey()
{
string secretKey = "";

// implement the way of querying
// for secret key

return secretKey;
}

// the generation of the output as query
// token for the trapdoor
public: string QueryToken()
{
string query_token = "";

// generate and build
// the query token for trapdoor

return query_token
}
}

class Search
{
// Step 5
// the algorithm from Search step (algorithm)
// are runned and invoked by the server
```

```
// the constructor of the Search class
public: Search() {}

public: string SearchQuery()
{
string query = "";

// take the search query

return query;
}

public: string Index()
{
string index = "";

// take the search query

return index;
}

public: string ReturnResult()
{
string result = "";
string setOfIdentifiers = "";

// based on the search query and index,
// get the set identifiers of the data items
setOfIdentifier = "query for identifiers";

// build the result. "#" is the separator for
// illustration purpose only
result = SearchQuery + "#" + Index;

return result;
}
}
```

```
class RevokeUser
{
// Step 6
// the algorithm from Search step (algorithm)
// are runned and invoked by the data owner

// constructor of RevokeUser class
public: RevokeUser(){}

// second constructor of the class
// this can be implemented as a
// solution for revoking a user
public: RevokeUser(string userID, string secretKeyDataOwner, string
secretKeyServer)
{
// implement the revoking process

// output the new key for data owner

// output the new key for server
}
}

public class Course
{
// the db_panel represents an instance of the
// file which contains classes for each of tables
// from the database
public: Database db_panel;

// Class Courses it is a generated class and assigned
// to the table Courses from the database
public: Courses c;

// student ID
string demoStudentID = “435663”;
```

```
// select the course ID based on the student
public: string GetCourse()
{
// select the courses for a
// specific user (student)
Course c = db_panel.GetCourse(student.Id);

return c;
}
}

class Student
{
public: string secretKey {get; set;}
public: int StudentId {get; set;}
public: string CourseID {get; set;}
public: string StudentName {get; set;}
public: string StudentIdentity {get; set;}
public: string StudentPersonalCode {get; set;}

public: void Student(string secret_key)
{
     secretKey = secret_key;
}
}

string queryKeywod =
    SecureSearch.GetPrefix("123456789");

string resultStudent = SecureSearch.GetStudent.StartsWith(searchPrefix);
```

Conclusion

In this chapter, we presented searchable encryption schemes and provided guidelines for a possible practical use that supports searchable encryption.

The potential of searchable encryption, which is a particular case of homomorphic encryption, is great in many domains of activity. In this chapter, we outlined the main components of searchable encryption schemes. If you're interested in more theoretical aspects for searchable encryption, any of the references provide a deeper view of SE. For some recent samples of pseudo-code, consult [11] or [12].

References

[1] E.J. Goh, "Secure indexes." IACR Cryptology ePrint Archive, 216. 2003.

[2] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search" in *International conference on the theory and applications of cryptographic techniques* (pp. 506-522). Springer; Berlin, Heidelberg. May, 2004.

[3] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing" in *2010 Proceedings IEEE INFOCOM* (pp. 1-5). IEEE. March, 2010.

[4] J. Bringer, H. Chabanne, and B. Kindarji, "Error-tolerant searchable encryption" in *2009 IEEE International Conference on Communications* (pp. 1-6). IEEE. June, 2009.

[5] J. Lai, X. Zhou, R.H. Deng, Y. Li, and K. Chen, "Expressive search on encrypted data" in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (pp. 243-252). May, 2013.

[6] R. Bost, "$\sum$ οφος: Forward secure searchable encryption" in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1143-1154). October, 2016.

[7] D.X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data" in *Proceeding 2000 IEEE Symposium on Security and Privacy*. S&P 2000 (pp. 44-55). IEEE. May, 2000.

[8] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption" in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1038-1055). January, 2018.

[9] C. Zuo, S.F. Sun, J.K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy" in *European Symposium on Research in Computer Security* (pp. 283-303). Springer, Cham. September, 2019.

[10] Crypteron documentation. Available online: www.crypteron.com/docs/.

[11] C. Ma, Y. Gu, and H. Li, "Practical Searchable Symmetric Encryption Supporting Conjunctive Queries without Keyword Pair Result Pattern Leakage."

[12] S. Fu, Q. Zhang, N. Jia, and M. Xu, "A Privacy-preserving Fuzzy Search Scheme Supporting Logic Query over Encrypted Cloud Data" in *Mobile Networks and Applications*, 1-12. 2020.

[13] Dan Boneh et al. "Public key encryption with keyword search" in *International conference on the theory and applications of cryptographic techniques*. Springer; Berlin, Heidelberg, 2004.

[14] Ernest Bonnah and Ju Shiguang, "Privacy Enhancement Scheme (PES) in a Blockchain-Edge Computing Environment" in *IEEE Access 2020*. October, 2019.

[15] Mohammad Shahriar Rahman et al, "Accountable cross-border data sharing using blockchain under relaxed trust assumption" in *IEEE Transactions on Engineering Management*. 2020.

[16] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A Survey of Provably Secure Searchable Encryption" in *ACM Comput. Surv.*, vol. 47, no. 2 (pp. 1–51). 2014.

[17] Prabhat Kumar Panda and Sudipta Chattopadhyay, "A secure mutual authentication protocol for IoT environment" in *Journal of Reliable Intelligent Environments* (pp. 1-16). 2020.

[18] Marius Iulian Mihailescu, Nita Stefania Loredana, and Pau Valentin Corneliu, "E-Learning System Framework using Elliptic Curve Cryptography and Searchable Encryption" in *Proceedings of International Scientific Conference for e-Learning and Software for Education, Volume 1* (pp 545-552). DOI: 10.12753/2066-026X-20-071. 2020.

CHAPTER 12

Homomorphic Encryption

Important types of encryption schemes are those that fall into the homomorphic encryption (HE) category, which allows calculations to be computed directly on the encrypted data, without needing a preceding decryption operation. The most important condition in homomorphic encryption is that the value achieved by decrypting the result obtained by applying the calculations over the encrypted data must be the same as the value achieved by applying the same calculations on the plain data. With these properties, the HE schemes are considered to have great potential because they enable third-party entities to apply functions (therefore, to apply algorithms) on the encrypted data, but without the need for any access of the plain data. In this way, the data is protected and secured while being processed. For a real-life example, suppose you are on vacation into a foreign city and you want to search the Internet, using your phone or another device, for local attractions such as museums, exhibitions, art galleries, and so on. Even this simple search on the Internet may reveal a lot of information about you, such as your exact location, your cultural interests, the time of the search query, and so on. If the search engine used a homomorphic approach, then nothing would be revealed to anyone including the search engine itself, because all of the information and even the search query would be encrypted. The results that you receive would be also encrypted, therefore only you could decrypt them. Homomorphic encryption has applications in many areas, such as finance/business, healthcare, and any domain that works with sensitive data. Further, some formal aspects of homomorphic encryption are given.

The function $g: A \to B$ is called *homomorphic over the operation* $*$ if the following condition is satisfied:

$$g(x_1) * g(x_2) = g(x_1 * x_2), \forall x_1, x_2 \in A$$

Remember that a general encryption system consists of the following algorithms: key generation, encryption, and decryption. Besides these three algorithms, the homomorphic encryption schemes have an additional algorithm called evaluation, which is usually

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_12

denoted with `Eval`*, which describes formally the most important rule mentioned above. The input and the output of the* `Eval` *algorithm are in an encrypted format.* In the `Eval` algorithm, the function g is applied over encrypted data c_1 and c_2, without accessing the plain data m_1 and m_2, and having the following property:

$$Dec\left(key_{priv}, Eval_g\left(key_{eval}, c_1, c_2\right)\right) = f\left(m_1, m_2\right)$$

In homomorphic encryption, only two operations are required to have homomorphic properties, namely addition and multiplication. This is due to the fact that an arbitrary function can be represented as a circuit using just gates corresponding to the addition operation (`OR` gate) and the multiplication operation (`AND` gate). The idea of homomorphic encryption started in the late 70s, when it was called *privacy homomorphism* [1]. Among the first encryption schemes to have homomorphic properties is the Unpadded RSA algorithm [2], in which the operation with homomorphic properties is the multiplication:

$$Encyption\left(m_1\right)\ \ Encryption\left(m_2\right) = m_1^e m_2^e \bmod n$$

$$= \left(m_1 m_2\right)^e \bmod n$$

$$= Encryption\left(m_1 \cdot m_2\right)$$

In the above computation, m_1, m_2 are two plain messages and *Encryption* is the encryption function.

The homomorphic encryption schemes can be categorized into three classes, as follows:

- **Partial homomorphic encryption (PHE)**: The schemes in this category support just one operation applied over encrypted data an unlimited number of times. Examples of PHE schemes are RSA [2], Goldwasser-Micali [3], and El-Gamal [4]. Most of the schemes from this category represent a basis for other homomorphic schemes.
- **Somewhat homomorphic encryption (SWHE)**: The schemes in this category support both operations applied on the encrypted data, but for a limited number of times. The encryption scheme from [5] is an example of SWHE.

- **Fully homomorphic encryption (FHE)**: The schemes in this category support both operations over encrypted data for an unlimited number of times. Fully homomorphic encryption is considered "cryptography's holy grail" or "the Swiss Army knife of cryptography" [6] due to its capability of enabling any computation over the encrypted data by any number of times. In 2009, the first FHE scheme [7] was proposed and the mathematical object used as the foundation is the ideal lattices. The scheme from [7] is very important in cryptography because it opened the way for the FHE schemes, and even though it is unpractical in the form in which it was proposed due to its complexity and abstraction, it represented a basis for subsequent schemes. In addition, in [7] a general framework for the FHE schemes was proposed.

Fully Homomorphic Encryption

In this section, fully homomorphic encryption (FHE) is explained in more detail because it represents an important topic of cryptography that can resolve many security concerns and issues. A particular model of quantum computation called *boson scattering* enables a quantum homomorphic encryption that provides theoretically limited security. By existing, this kind of scheme makes us wonder if quantum methods can generate theoretically secure FHE schemes. In [25] the authors prove that quantum techniques do not enable efficient theoretically secure FHE that hides completely the plaintext.

As mentioned in the previous section, the first FHE scheme was proposed by Craig Gentry in 2009 and the mathematical object that represents the foundation is the ideal lattices with the *hardness assumption* (problems regarding a topic that cannot be solved in an efficient time, i.e in polynomial time) called the *ideal coset problem*. Following Gentry's scheme, there were proposed a large number of FHE schemes based on different mathematical techniques. A subsequent work is [8], in which the FHE scheme uses integer arithmetic. However, the noise introduced in the schemes from [7] and [8] grows quickly, representing a drawback because it has a high effect over the applicability and security, thus the homomorphic capabilities are restricted. Due to the noise growth, the decryption cannot be made after some point.

In the second generation of the FHE schemes that include works such as [9] and [10], the noise is handled more efficiently, which means improved performance and powerful

security under various hardness assumptions. The *leveled encryption schemes* and *bootstrappable encryption schemes* are results of this generation. The first ones evaluate the circuits with a given polynomial depth, while the second ones can be modified to become FHE schemes. If an encryption scheme has the capability of evaluating its decryption circuit and additionally one NAND gate, then it is a bootstrappable encryption scheme.

The third generation of FHE schemes is opened by the work of [11], which uses a new technique to handle the noise. The schemes of the third generation are less performant than those from the second generation, but their hardness assumptions can be weaker. The basis for many schemes in this generation is asymmetric multiplication. That is, considering two encrypted texts c_1, c_2, the product $c_1 \bullet c_2$ is different from the product $c_2 \bullet c_1$, although both products encrypt the same product $b_1 \bullet b_2$ of the plain texts b_1 and b_2.

FHE can be used in many areas of cryptography, such as

- **Outsourcing**: The private data can be kept safe if it is stored in third-party storage or analyzed by third-party entities. A classic example for this area is that of a company that stores its data in cloud storage. Before uploading the data in the cloud, the owner needs to encrypt it. FHE would be useful in such scenarios because the cloud provider could analyze the data from the company in an encrypted format, without accessing the plain data. Moreover, the result of the computations would be sent by the cloud provider in the encrypted format to the data owner, where it would be decrypted only by the decryption key's owner.
- **Private information retrieval (PIR) or private queries**: PIR and private queries are useful when a database is queried or an application uses a search engine. Another scenario is when a client wants to send a query to a database server, but the client wants the server to learn nothing about its query. The solution is the following: the client encrypts the query and sends it to the server and then the server applies the encrypted query over encrypted data and responds with the encrypted result.

- **General computations between two entities** (two-party computations): Consider two parties A and B. Each of them owns a secret input, x, and y, respectively, and a common function F known by both. To apply the function F over its private input x, the party A computes `r = F(x, y)`. From here, A learns only the value of r and learns nothing about y. On the other hand, B learns nothing about x or r. This is the same as B computing `Fy(x)` in the semi-honest model, where A encrypts x and sends to B because the semantic security assures the fact that B will learn nothing about the plain value corresponding to x. In such situations, using FHE would simplify things, because A would just apply F as `F(x,y)`, and achieve the result in an encrypted format, but it would need and learn nothing else because everything is encrypted, including F.

Practical Example of Using FHE

There are more libraries for C++ that implement fully homomorphic encryption. Some well-known libraries for C++ FHE are the following:

- HElib [12], developed at IBM, implements the schemes BFV (Brakerski/Fan-Vercauteren) [17] and CKKS (Cheon-Kim-Kim-Song) [18] and it can be used in Linux and MacOS distributions.
- TFHE [13] implements the scheme proposed in [15] and it can be used with Linux distributions. In the same paper, the library is described.
- PALISADE [14] implements the BGV (Brakerski-Gentry-Vaikuntanathan) [16], BFV [17], CKKS [18], and FHEW schemes [19] and a more secure version of the TFHE scheme [13], including bootstrapping. It is supported on Linux, Windows, and macOS distributions.
- SEAL [20]-[23] implements the BFV [17] and CKKS [18] schemes and it can be used with .Net or C++. In addition, the SEAL library can be used in Windows, Linux, or MacOS environments.

In this section, we'll use SEAL library to demonstrate an FHE example. The Seal library implements BFV [12] and CKKS [13] encryption schemes.

In [12], the set of the polynomials with a maximum degree n and the coefficients computed modulo t is used in the definition of the encryption function. The formal representation of this set is $R_t = Z_t[x]/(x^n + 1)$. The encrypted text is from the R_q set, where the polynomials have coefficients modulo q. The addition and the multiplication are the homomorphic operations in this encryption scheme, preserving the ring structure of R_t. The value that needs to be encrypted using BFV schemes first needs to be brought to a polynomial form accepted by the structure R_t. In [12] the encryption scheme includes the following algorithms: `SecretKeyGen` (the security parameter is used to generate the secret key), `PublicKeyGen` (the secret key is used to generate the public key), `EvaluationKeyGen` (the secret key is used to generate the evaluation key), `Encrypt` (the plain value is encrypted using the public key), `Decrypt` (the encrypted value is decrypted using the secret key), `Add` (performs the addition between two encrypted values), and `Multiply` (performs the multiplication between two encrypted values). Keep in mind that the results of both operations, namely addition and multiplication, have a form that is compatible with the structure R_q. For more details and a formal description of this encryption scheme, you can consult [12].

While [12] provides a way to apply modular arithmetic over integers, in [13] the authors provide ways to apply it over real numbers and complex numbers, too. Anyway, in [13] the results are approximate, but the techniques are among the best to sum up real numbers in an encrypted format, to apply machine learning algorithms on encrypted data, or to compute distanced between encrypted locations.

Before using SEAL library, some preparatory steps are needed, which are described below.

First, install a version of Visual Studio 2019. The community version, which is free, can be found at `https://visualstudio.microsoft.com/vs/community`. Make sure that the C++ components (under *Desktop development with C++*) are checked to be installed. Then download Git from `https://git-scm.com/download/win` and install it via the installations steps with the default values. After these programs are set, the SEAL library can be downloaded from the GitHub repository: `https://github.com/microsoft/SEAL` (at the moment of writing this book, the latest version of SEAL is 3.5.6). After downloading the source code, extract the zip file. We left the default name `Seal-master` and extracted it in the path `C:\libs`. Open the `Seal-master` folder and then open the `SEAL.sln` file using Visual Studio. The structure of the solution should be as in Figure 12-1.

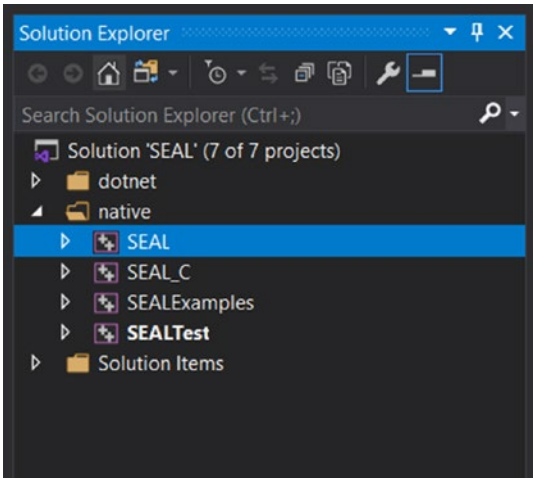


Figure 12-1. *The structure of the SEAL.sln solution*

The folder used for C++ development is the `native` folder from the solution. To use SEAL library in your own C++ application, you need first to generate the `seal.lib` library. To do this, you need to build the SEAL project from Figure 12-1. From the Toolbar, pick Release configuration and x64 platform (Figure 12-2(a) and 12-2(b)), then right-click the SEAL project and choose Build. The Release configuration is needed because things go faster than in the Debug configuration and you actually just need to generate the `seal.lib`, not debug it. Right-click the SEAL project under the native folder and choose Build or Rebuild.

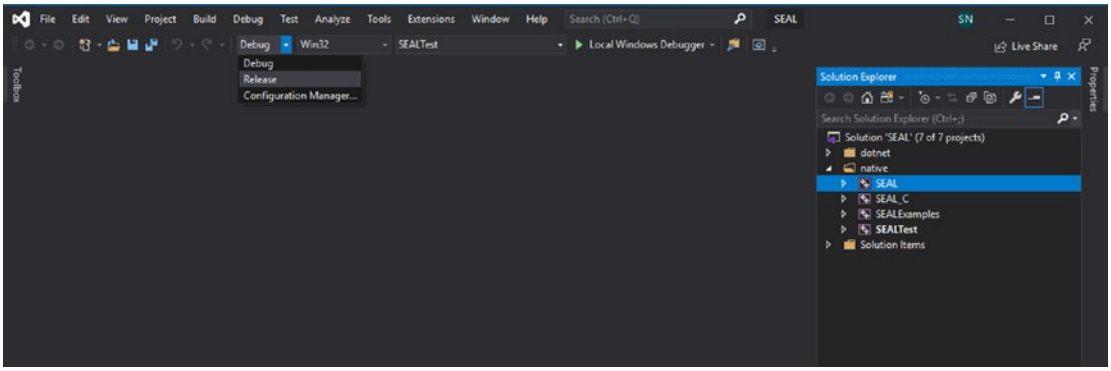


Figure 12-2(a). *Choosing the configuration for SEAL building*

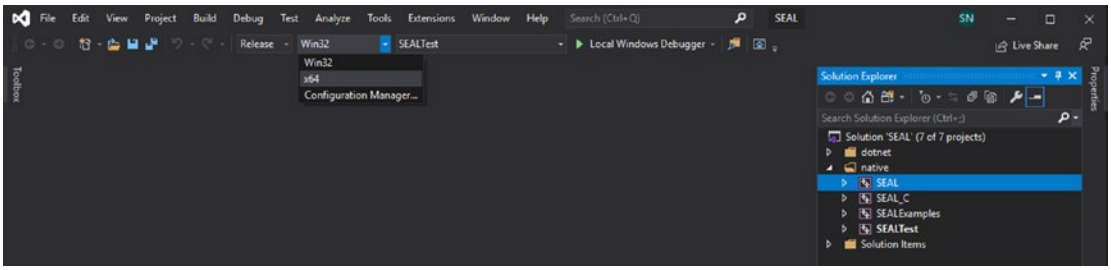


Figure 12-2(b). *Choosing the platform for SEAL building*

If everything works properly, a similar message as in Listing 12-1 should be obtained.

Listing 12-1. The Result of Building the SEAL Project

```
...
1>-- Configuring done
1>-- Generating done
1>-- Build files have been written to: C:/libs/SEAL-master/.config/16.0/x64
1>SEAL.vcxproj -> C:\libs\SEAL-master\native\src\..\..\lib\x64\Release\
seal.lib
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Checking the path `C:\libs\SEAL-master\lib\x64\Release`, you should find the library `seal.lib`. Now you are ready to create your own application that uses FHE. In Visual Studio, create an empty project of type `Console App` with C++ called `SealCPPExample` and add under the Source Files folder a `cpp` file called `SealExample.cpp`. Here, add an empty main function, as in Listing 12-2.

Listing 12-2. The Initial Main Function

```
int main()
{
    return 0;
}
```

Further, the application needs to be prepared for using SEAL library as described as below. First, right-click the `SealCPPExample` solution and go to Properties. Here, make sure All Configurations and All Platforms are selected (Figure 12-3).

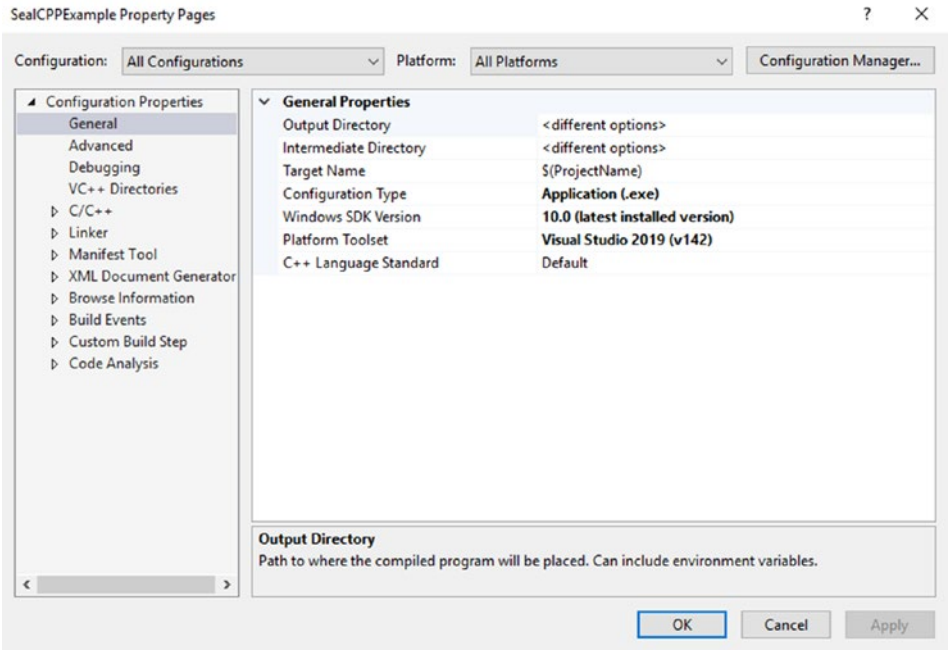


Figure 12-3. *Settings for using SEAL library (1)*

Then, under C/C++ ➤ General ➤ Additional Include Directories**,** add the path where sources were generated (in our example, the path is `C:\libs\SEAL-master\native\src`; see Figure 12-4).

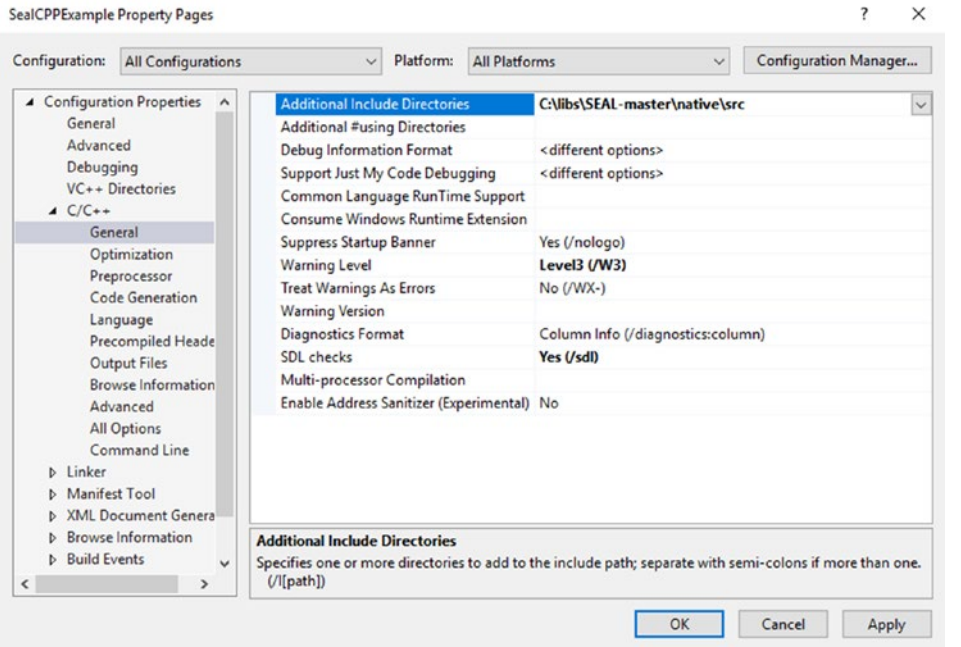


Figure 12-4. *Settings for using SEAL library (2)*

Finally, to include `seal.lib`: under Linker ➤ Additional Library Directories, add the path to `seal.lib` (in our example, the path is `C:\libs\SEAL-master\lib\$(Platform)\$(Configuration)`; see Figure 12-5(a)). Note that the $(Platform) for our example is x64 and the $(Configuration) is Release. The final step is to add `seal.lib` to Linker ➤ Input ➤ Additional Dependencies (Figure 12-5(b)).

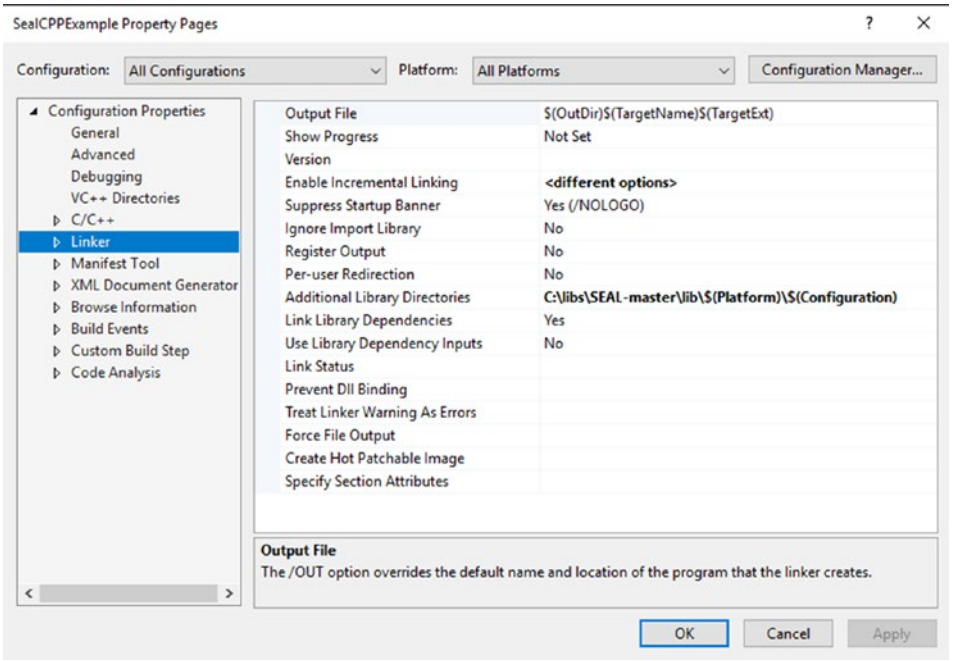


Figure 12-5(a). *Settings for using SEAL library (3)*

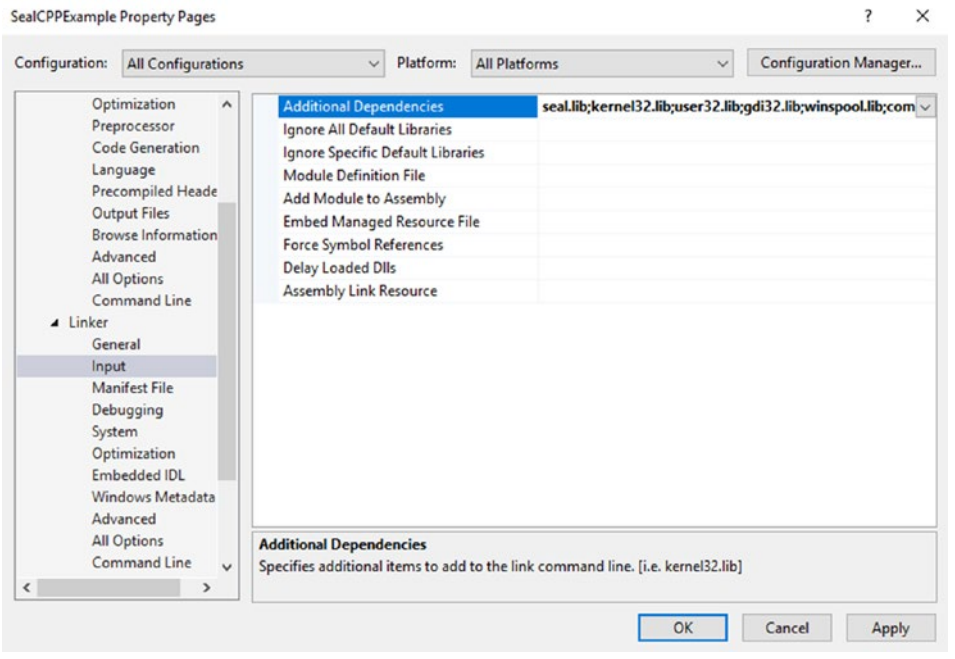


Figure 12-5(b). *Settings for using SEAL library (4)*

To make sure that SEAL was added properly, just add the line from Listing 12-3 in the main function and then build the solution. Do not forget to choose the Release configuration and x64 platform, and then right-click the solution and choose Build.

Listing 12-3. Checking If SEAL Has Been Added Properly

```
EncryptionParameters BFV_parameters (scheme_type::BFV);
```

If a success message is returned, then you can proceed further; otherwise, if an error message similar to `'for_each_n': is not a member of 'std'` is returned, then one more step is needed: change the C++ Language Standard under C/C++ > Language from Default to ISO C++17 Standard (/std:c++17).

Create a function called `seal_example_bfv`, in which functionalities provided by SEAL library for BFV encryption scheme are added. In the first place, the encryption parameters should be added: the degree of the polynomials from the ring (`n`), the modulus for the coefficients of the plaintext (`t`), and the modulus for the coefficients of the encrypted text (`q`). To use the `SEAL` functionalities, the libraries from Listing 12-4 should be added. The application is notified that the BFV scheme is used and instantiates the parameters using the line of code from Listing 12-5.

Listing 12-4. The Libraries Included for `SEAL`

```
#pragma once

#include "seal/seal.h"
#include <iostream>
#include <algorithm>
#include <chrono>
#include <cstddef>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <memory>
#include <mutex>
#include <numeric>
#include <random>
#include <sstream>
```

```
#include <string>
#include <thread>
#include <vector>

using namespace std;
using namespace seal;
```

Listing 12-5. Instantiating the BFV Parameters

```
void seal_example_bfv()
{
    EncryptionParameters BFV_parameters(scheme_type::BFV);}
```

After instantiating the BFV parameters, they should receive each a value. *The degree of the polynomial modulus* is a power of 2 and represents a degree of a cyclotomic polynomial[1]. The values that are recommended for it are {1024, 2048, 4096, 8192, 16384, 32768}. With a higher value for the polynomial degree, more complex computations on the encrypted data can be made, but the drawback is that the performance decreases. A fair value is 4096, allowing for an acceptable number of computations with a good performance, therefore this value was chosen for our application. *The modulus for the coefficients of the plaintext* is in general a positive integer. The value for this parameter is a power of two in our example. Depending on the purpose of the application, the modulus can be a prime number. The modulus for the coefficient of the plaintext is used to provide the size in bits for the plain data and to establish limits for consumption in the multiplication operation. The last parameter is *the modulus for the coefficients of the encrypted text*, which represents a large integer value. The value for this modulus should be represented as the product of prime numbers. When a larger value is chosen, more computations over the encrypted data can be made. However, there is a relation between the degree of the polynomial modulus and the size in bits of the modulus for the coefficients of the encrypted text, therefore a 4096 value corresponds to the value 109. Comprehensive explanations for the scheme's parameters can be found in [20] and [21].

The other functionality that needs a few words is *the noise budget*, representing a number of bits. On short, the initial noise budget is set depending on the encryption parameters and the rate at which the homomorphic operations (addition and

[1] https://en.wikipedia.org/wiki/Cyclotomic_polynomial

multiplication) consume it. The parameter that has the highest influence in setting the noise budget is the coefficient modulus. When a higher value is picked, the budget is higher. When the noise budget for an encryption text becomes 0, then the decryption of the encrypted text cannot be made anymore, because the noise it contains has a value too large.

With these brief descriptions, the parameters can be initialized using the lines of code from Listing 12-6, added in the function `seal_example_bfv`.

Listing 12-6. Initialization of the BFV Parameters

```
size_t polynomial_degree = 4096;
BFV_parameters.set_poly_modulus_degree(polynomial_degree);
    BFV_parameters.set_coeff_modulus(CoeffModulus::BFVDefault(polynomial_
    degree));
BFV_parameters.set_plain_modulus(1024);
```

The SEAL context checks the correctness of the parameters:

```
auto seal_context = SEALContext::Create(BFV_parameters);
```

Further, the classes for the BFV encryption scheme need to be instantiated, as shown in Listing 12-7 (code added in function `seal_example_bfv`).

Listing 12-7. Instantiating the Classes for the BFV Encryption Scheme

```
KeyGenerator keygen(seal_context);
PublicKey encryption_key = keygen.public_key();
SecretKey decryption_key = keygen.secret_key();
Encryptor bfv_encrypt(seal_context, encryption_key);
Evaluator bfv_evaluate(seal_context);
Decryptor bfv_decrypt(seal_context, decryption_key);
```

In the following, for this example, the polynomial $p(x) = 3x^4 + 6x^3 + 9x^2 + 12x + 6$ will be evaluated for $x = 3$. For a quick check, you can use the value $x = 3$ to encrypt and then decrypt it. Listing 12-8 shows this process and shows some metrics (code added in `seal_example_bfv` function).

Listing 12-8. Encrypting and Decrypting x=3

```
int value_x = 3;
Plaintext x_plaintext(to_string(value_x));
cout << "The value x = " + to_string(value_x)
     + " is expressed as a plaintext polynomial 0x"
     + x_plaintext.to_string() + "." << endl;

Ciphertext x_ciphertext;
cout << "Encrypting x_plaintext to x_ciphertext..." << endl;
bfv_encrypt.encrypt(x_plaintext, x_ciphertext);

cout << "    - the size of the x_ciphertext (freshly
        encrypted) is : "
     << x_ciphertext.size() << endl;
cout << "    - the noise budget for x_ciphertext is : "
     << bfv_decrypt.invariant_noise_budget(x_ciphertext)
     << " bits" << endl;
Plaintext value_x_decrypted;
cout << "    - decryption of x_encrypted: ";
bfv_decrypt.decrypt(x_ciphertext, value_x_decrypted);
cout << "0x" << value_x_decrypted.to_string() << endl;
```

Next, call `seal_example_bfv` in the main function as follows:

```
int main()
{
    seal_example_bfv();
    return 0;
}
```

To run the application, do not forget to choose Release configuration and x64 platform, and then press CTRL+F5. The result should be similar to Listing 12-9 and Figure 12-6.

Listing 12-9. The Output for the Encryption, Decryption and Metrics

```
The value x = 3 is expressed as a plaintext polynomial 0x3.
Encrypting x_plaintext to x_ciphertext...
    - the size of the x_ciphertext (freshly encrypted) is : 2
    - the noise budget for x_ciphertext is : 55 bits
    - decryption of x_encrypted: 0x3
```

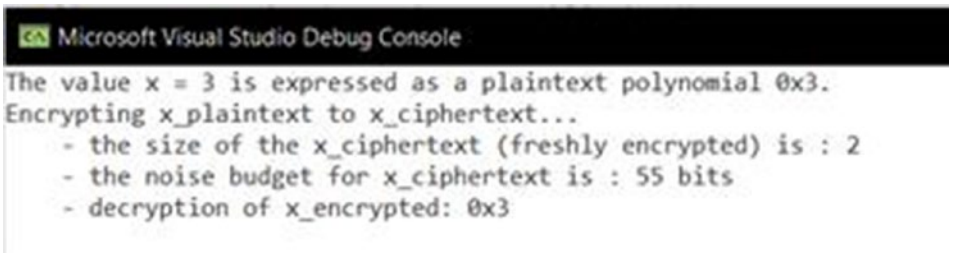


Figure 12-6. *The output for the encryption, decryption and metrics*

The `Plaintext` constructor converts the plain values to polynomials that have a degree lower than the modulus polynomial, for which the coefficients are represented as hexadecimal values. In `SEAL`, the encrypted text is represented as two or more polynomials with coefficients in the form of inter values modulo, the result of the multiplication of the prime numbers from `CoeffModulus` representation. The object `x_ciphertext` instantiates the class `Ciphertext` and receives the value of the encryption of `x_plaintext` through calling the `encrypt` method of the object `bfv_encrypt`. This method takes two parameters, namely the object that needs to be encrypted (`x_plaintext`) and the object in which the encryption of the first parameter should be put (`x_ciphertext`). The number of the polynomials gives the size of the encrypted text; a fresh encrypted text has the size 2, which is returned by the `size()` method of the object `x_ciphertext.` The noise budget is computed by the `invariant_noise_budget()` method of the `bfv_encrypt` object, which takes as a parameter the object `x_ciphertext.` The `invariant_noise_budget()` is implemented into the `Decryptor` class because it shows if the decryption will work at some point in the computations. To decrypt the encrypted value obtained, use the `decrypt` method, called by the `bfv_decrypt` object. The decryption works because the value `0x3` in hexadecimal representation means `3`.

For optimizations, the recommendation is that the polynomials be brought to a form that includes as few possible multiplication operations, because they are costly operations that will decrease the noise budget fast. Therefore, $p(x)$ may be factorized as

$p(x) = 3(x^2 + 2)(x + 1)^2$, which means you will evaluate first $(x^2 + 2)$, then $(x + 1)^2$ and then you will multiply the result between them and with 3. To compute $(x^2 + 2)$, proceed as presented in Listing 12-10 (code added in `seal_example_bfv` function).

Listing 12-10. Computing $(x^2 + 2)$

```
cout << "Computing (x^2+2)..." << endl;
Ciphertext square_x_plus_two;
bfv_evaluate.square(x_ciphertext, square_x_plus_two);
Plaintext plain_value_two("2");
bfv_evaluate.add_plain_inplace(square_x_plus_two,
             plain_value_two);

cout << "    - the size of the square_x_plus_two is: "
     << square_x_plus_two.size() << endl;
cout << "    - the noise budget for square_x_plus_two is: "
     << bfv_decrypt.invariant_noise_budget(square_x_plus_two)
     << " bits" << endl;

Plaintext decrypted_result;
cout << "    - decryption of square_x_plus_two: ";
bfv_decrypt.decrypt(square_x_plus_two, decrypted_result);
cout << "0x" << decrypted_result.to_string() << endl;
```

After running the application, you obtain the result from Listing 12-11 and Figure 12-7.

Listing 12-11. The Result of Computing $(x^2 + 2)$

```
The value x = 3 is expressed as a plaintext polynomial 0x3.
Encrypting x_plaintext to x_ciphertext...
    - the size of the x_ciphertext (freshly encrypted) is : 2
    - the noise budget for x_ciphertext is : 55 bits
    - decryption of x_encrypted: 0x3
Computing (x^2+2)...
    - the size of the square_x_plus_two is: 3
    - the noise budget for square_x_plus_two is: 33 bits
    - decryption of square_x_plus_two: 0xB
```

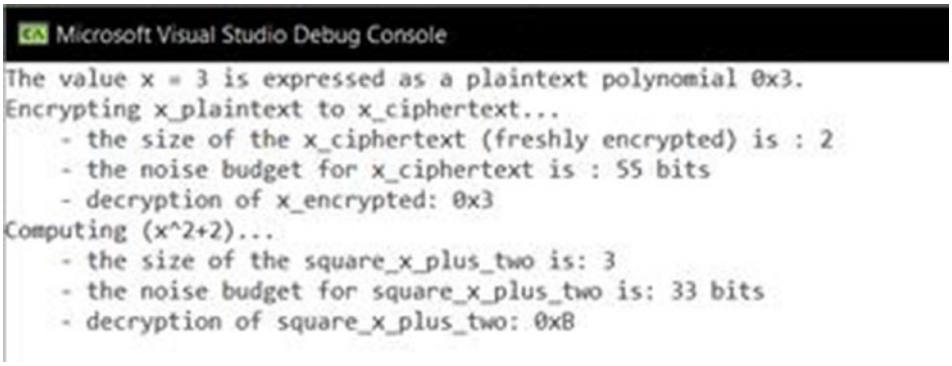


Figure 12-7. *The result of computing* $(x^2 + 2)$

For checking, if you calculate $3^2 + 2$ you obtain `11`, whose hexadecimal representation is `0xB`; the noise budget is greater than 0, which means the decryption can be made. Observe that the `bfv_evaluate` object allows applying operations directly over the encrypted data. The collector variable for this example is `square_x_plus_two`. First, this variable keeps the encrypted value raised at power 2, i.e. x^2, using the method `square()`. Further, you add plain value 2, through the method `add_plain_inplace()`, which gives $x^2 + 1$. Remember that in this example $x = 3$. The methods `square()` and `add_plain_inplace()` methods have two parameters, namely a source and a destination.

Similarly, you compute $(x + 1)^2$ using as a collector variable `x_plus_one_square` (see Listing 12-12).

Listing 12-12. Computing $(x + 1)^2$

```
cout << "Computing (x+1)^2..." << endl;
Ciphertext x_plus_one_square;
Plaintext plain_value_one("1");
bfv_evaluate.add_plain(x_ciphertext, plain_value_one,
            x_plus_one_square);
bfv_evaluate.square_inplace(x_plus_one_square);
cout << "    - the size of x_plus_one_square is: "
     << x_plus_one_square.size() << endl;
cout << "    - the noise budget in x_plus_one_square is: "
     << bfv_decrypt.invariant_noise_budget(x_plus_one_square)
     << " bits" << endl;
```

```
cout << "    - decryption of x_plus_one_square: ";
bfv_decrypt.decrypt(x_plus_one_square, decrypted_result);
cout << "0x" << decrypted_result.to_string() << endl;
```

And you obtain after running the application the results shown in Listing 12-13 and Figure 12-8.

Listing 12-13. The Result of Computing $(x + 1)^2$

```
The value x = 3 is expressed as a plaintext polynomial 0x3.
Encrypting x_plaintext to x_ciphertext...
    - the size of the x_ciphertext (freshly encrypted) is : 2
    - the noise budget for x_ciphertext is : 55 bits
    - decryption of x_encrypted: 0x3
Computing (x^2+2)...
    - the size of the square_x_plus_two is: 3
    - the noise budget for square_x_plus_two is: 33 bits
    - decryption of square_x_plus_two: 0xB
Computing (x+1)^2...
    - the size of x_plus_one_square is: 3
    - the noise budget in x_plus_one_square is: 33 bits
    - decryption of x_plus_one_square: 0x10
```

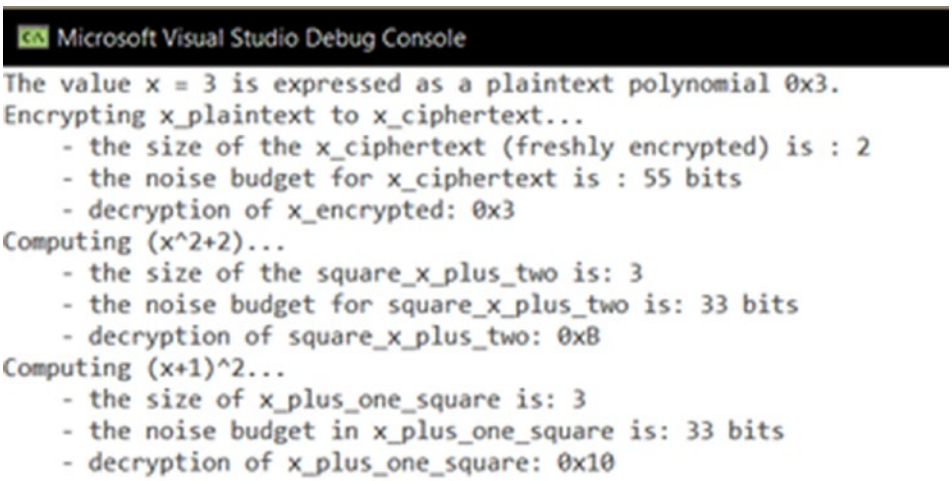


Figure 12-8. *The result of computing* $(x + 1)^2$

Indeed, if you compute $(3 + 1)^2$ you get 10, whose hexadecimal representation is 0x10; the noise budget is greater than 0, so the decryption still works.

The final result of $3(x^2 + 2)(x + 1)^2$ is collected into encryptedOutcome variable (see Listing 12-14).

Listing 12-14. Computing $3(x^2 + 2)(x + 1)^2$

```
cout << "Compute [3(x^2+2)(x+1)^2]." << endl;
Ciphertext enc_result;
Plaintext plain_value_three("3");
    bfv_evaluate.multiply_plain_inplace(square_x_plus_two,
       plain_value_three);
bfv_evaluate.multiply(square_x_plus_two, x_plus_one_square,
            enc_result);
cout << "    - the size of encrypted_result: "
     << enc_result.size() << endl;
cout << "    - the noise budget for encrypted_result: "
     << bfv_decrypt.invariant_noise_budget(enc_result)
     << " bits" << endl;
cout << "NOTE: If the noise budget is zero, the decryption can be
incorrect." << endl;
cout << "    - decryption of enc_result: ";
bfv_decrypt.decrypt(enc_result, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << endl;
```

And you obtain after running the application the results shown in Listing 12-15 and Figure 12-9.

Listing 12-15. The Output of Computing $3(x^2 + 2)(x + 1)^2$

```
The value x = 3 is expressed as a plaintext polynomial 0x3.
Encrypting x_plaintext to x_ciphertext...
    - the size of the x_ciphertext (freshly encrypted) is : 2
    - the noise budget for x_ciphertext is : 55 bits
    - decryption of x_encrypted: 0x3
Computing (x^2+2)...
    - the size of the square_x_plus_two is: 3
```

```
    - the noise budget for square_x_plus_two is: 33 bits
    - decryption of square_x_plus_two: 0xB
Computing (x+1)^2...
    - the size of x_plus_one_square is: 3
    - the noise budget in x_plus_one_square is: 33 bits
    - decryption of x_plus_one_square: 0x10
Compute [3(x^2+2)(x+1)^2].
    - the size of encrypted_result: 5
    - the noise budget for encrypted_result: 4 bits
NOTE: If the noise budget is zero, the decryption can be incorrect.
    - decryption of enc_result: 0x210
```

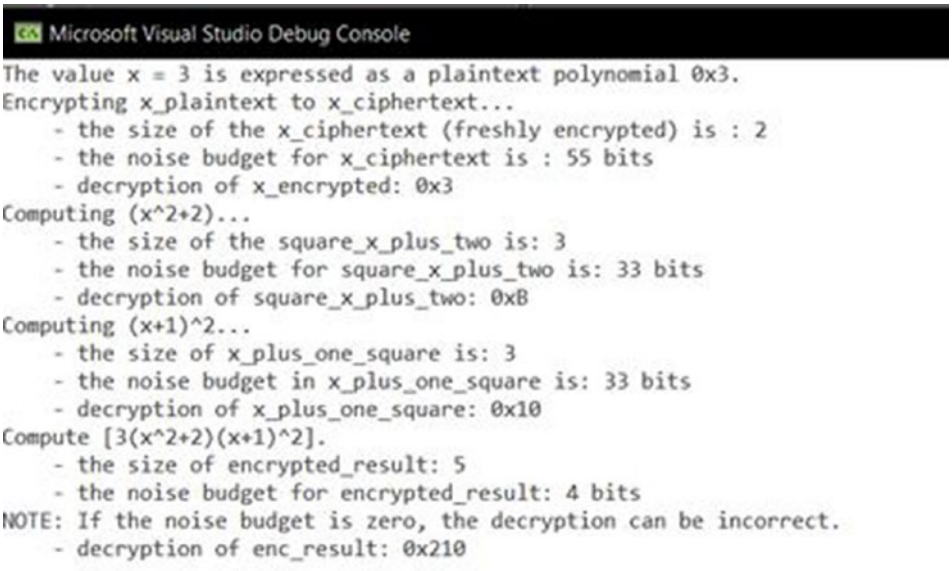


Figure 12-9. *The output of computing* $3(x^2 + 2)(x + 1)^2$

Indeed, if you compute $3(3^2 + 2)(3 + 1)^2$ you get `528`. Do not forget that the plaintext modulus is `1024`, so `528 mod 1024 = 528`, which has the `0x210` hexadecimal representation. The noise budget is greater than 0, which allowed you to decrypt the final encrypted result.

Putting all together, see the code in the `SealExample.cpp` file (Listing 12-16).

Listing 12-16. The Entire Code

```
#pragma once

#include "seal/seal.h"
#include <iostream>
#include <algorithm>
#include <chrono>
#include <cstddef>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <memory>
#include <mutex>
#include <numeric>
#include <random>
#include <sstream>
#include <string>
#include <thread>
#include <vector>

using namespace std;
using namespace seal;

void seal_example_bfv()
{
    EncryptionParameters BFV_parameters(scheme_type::BFV);

    size_t polynomial_degree = 4096;
    BFV_parameters.set_poly_modulus_degree(polynomial_degree);
    BFV_parameters.set_coeff_modulus(CoeffModulus::BFVDefault(polynomial_
    degree));
    BFV_parameters.set_plain_modulus(1024);

    auto seal_context = SEALContext::Create(BFV_parameters);
```

```
KeyGenerator keygen(seal_context);
PublicKey encryption_key = keygen.public_key();
SecretKey decryption_key = keygen.secret_key();
Encryptor bfv_encrypt(seal_context, encryption_key);
Evaluator bfv_evaluate(seal_context);
Decryptor bfv_decrypt(seal_context, decryption_key);

int value_x = 3;
Plaintext x_plaintext(to_string(value_x));
cout << "The value x = " + to_string(value_x) + " is expressed as a
plaintext polynomial 0x" + x_plaintext.to_string() + "." << endl;

Ciphertext x_ciphertext;
cout << "Encrypting x_plaintext to x_ciphertext..." << endl;
bfv_encrypt.encrypt(x_plaintext, x_ciphertext);

cout << "    - the size of the x_ciphertext (freshly encrypted) is : "
<< x_ciphertext.size() << endl;
cout << "    - the noise budget for x_ciphertext is : " << bfv_decrypt.
invariant_noise_budget(x_ciphertext) << " bits"
    << endl;

Plaintext value_x_decrypted;
cout << "    - decryption of x_encrypted: ";
bfv_decrypt.decrypt(x_ciphertext, value_x_decrypted);
cout << "0x" << value_x_decrypted.to_string() << endl;

cout << "Computing (x^2+2)..." << endl;
Ciphertext square_x_plus_two;
bfv_evaluate.square(x_ciphertext, square_x_plus_two);
Plaintext plain_value_two("2");
bfv_evaluate.add_plain_inplace(square_x_plus_two, plain_value_two);

cout << "    - the size of the square_x_plus_two is: " << square_x_
plus_two.size() << endl;
cout << "    - the noise budget for square_x_plus_two is: " << bfv_
decrypt.invariant_noise_budget(square_x_plus_two) << " bits"
    << endl;
```

```
    Plaintext decrypted_result;
    cout << "    - decryption of square_x_plus_two: ";
    bfv_decrypt.decrypt(square_x_plus_two, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << endl;

    cout << "Computing (x+1)^2..." << endl;
    Ciphertext x_plus_one_square;
    Plaintext plain_value_one("1");
    bfv_evaluate.add_plain(x_ciphertext, plain_value_one, x_plus_one_square);
    bfv_evaluate.square_inplace(x_plus_one_square);
    cout << "    - the size of x_plus_one_square is: " << x_plus_one_
    square.size() << endl;
    cout << "    - the noise budget in x_plus_one_square is: " << bfv_
    decrypt.invariant_noise_budget(x_plus_one_square) << " bits"
        << endl;
    cout << "    - decryption of x_plus_one_square: ";
    bfv_decrypt.decrypt(x_plus_one_square, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << endl;

    cout << "Compute [3(x^2+2)(x+1)^2]." << endl;
    Ciphertext enc_result;
    Plaintext plain_value_three("3");
    bfv_evaluate.multiply_plain_inplace(square_x_plus_two, plain_value_
    three);
    bfv_evaluate.multiply(square_x_plus_two, x_plus_one_square, enc_result);
    cout << "    - the size of encrypted_result: " << enc_result.size() <<
    endl;
    cout << "    - the noise budget for encrypted_result: " << bfv_decrypt.
    invariant_noise_budget(enc_result) << " bits"
        << endl;
    cout << "NOTE: If the noise budget is zero, the decryption can be
    incorrect." << endl;
    cout << "    - decryption of enc_result: ";
    bfv_decrypt.decrypt(enc_result, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << endl;
}
```

```
int main()
{
    seal_example_bfv();
    return 0;
}
```

In this section, we provided an easy example of how the SEAL library can be used with C++ on a Windows distribution. However, real-life applications are much more complex, which raises the need to handle more complex functions and algorithms.

The SEAL library can be very useful, and its big advantage is that it does not depend on other external libraries. When the applications work with the exact values of integers, the BFV encryption scheme implemented in the SEAL library is great. If the application needs to work with real or complex numbers, the CKKS encryption scheme is the better choice, which is also implemented in the SEAL library.

Conclusion

In this chapter,

- You learned what homomorphic encryption is and the types of homomorphic encryption.
- You got a deeper view of a fully homomorphic encryption and you saw why it is so important.
- You used Microsoft's SEAL library, which implements the BFV encryption scheme, on a simple example with a polynomial evaluation.

References

[1] R.L. Rivest, L. Adleman, and M.L. Dertouzos, "On data banks and privacy homomorphisms" *in Foundations of Secure Computation*, 4(11) (pp. 169-180). 1978.

[2] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems" in *Communications of the ACM 21.2* (pp. 120-126). 1978.

[3] Shafi Goldwasser and Silvio Micali, "Probabilistic encryption: how to play mental poker keeping secret all partial information" in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. 1982.

[4] Taher ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms" in *IEEE Transactions on Information Theory* 31.4 (pp. 469-472). 1985.

[5] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim, "Evaluating 2-DNF formulas on ciphertexts" in *Theory of Cryptography Conference*. Springer; Berlin, Heidelberg. 2005.

[6] B. Barak and Z. Brakerski, "The Swiss Army knife of cryptography", http://windowsontheory.org/2012/05/01/the-swiss-army-knife-of-cryptography/, 2012.

[7] Craig Gentry, "Fully homomorphic encryption using ideal lattices" in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. 2009.

[8] Marten Van Dijk et al, "Fully homomorphic encryption over the integers" in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer; Berlin, Heidelberg, 2010.

[9] Zvika Brakerski and Vinod Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE" in *SIAM Journal on Computing* 43.2 (pp. 831-871). 2014.

[10] Craig Gentry, Z. Brakerski, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping" in *Security 111.111* (pp. 1-12). 2011.

[11] Craig Gentry, Amit Sahai, and Brent Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based" in *Annual Cryptology Conference*. Springer; Berlin, Heidelberg. 2013.

[12] GitHub: HElib. Available online: https://github.com/homenc/HElib.

[13] TFHE: Fast Fully Homomorphic Encryption over the Torus. Available online: https://tfhe.github.io/tfhe.

[14] PALISADE Homomorphic Encryption Software Library. Available online: https://palisade-crypto.org.

[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds" in *International Conference on the Theory and Application of Cryptology and Information Security* (pp. 3-33). Springer; Berlin, Heidelberg. December, 2016.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping" in *ACM Transactions on Computation Theory (TOCT)*, 6(3) (pp. 1-36). 2014.

[17] Junfeng Fan and Frederik Vercauteren, "Somewhat Practical Fully Homomorphic Encryption" in *IACR Cryptology ePrint Archive 2012* (pp. 144). 2012.

[18] Jung Hee Cheon et al, "Homomorphic encryption for arithmetic of approximate numbers" in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, Cham, 2017.

[19] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second" in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 617-640). Springer; Berlin, Heidelberg. April, 2015.

[20] Hao Chen, Kim Laine, and Rachel Player, "Simple encrypted arithmetic library-SEAL v2.1" in *International Conference on Financial Cryptography and Data Security*. Springer, Cham, 2017.

[21] Kim Laine, "Simple encrypted arithmetic library 2.3.1." Microsoft Research. www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf. 2017.

[22] Microsoft SEAL. Available online: www.microsoft.com/en-us/research/project/microsoft-seal/.

[23] GitHub: microsoft/SEAL. Available online: https://github.com/Microsoft/SEAL.

[24] Martin R. Albrecht, "On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL" in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Cham, 2017.

[25] L. Yu, C.A. Pérez-Delgado, and J.F. Fitzsimons, "Limitations on information-theoretically-secure quantum homomorphic encryption" in *Physical Review A*, 90(5), 050303. 2014.

CHAPTER 13

Ring Learning with Errors Cryptography

The topic of this chapter is Ring Learning with Errors cryptography (RLWE). It's one of the most important and challenging techniques to use to develop secure and complex applications and systems.

The Learning with Errors (LWE) problem was introduced in 2005 through the work [4] by Oded Regev. Since then, it has proved its potential to be a basis for the future of cryptography and its capability to generate complex cryptographic structures. LWE and related topics are widely used in lattice-based cryptography. You can find comprehensive studies and surveys and deep formal aspects in the works [5, 6, 7, 8].

LWE is a difficult computation problem (therefore, a hardness assumption in cryptography) that is the formal foundation for cryptographic algorithms and constructions. One such cryptographic construction is *NewHope* [9], which is an encapsulation method for post-quantum keys. The purpose of NewHope is to protect against cryptanalysis attacks launched on quantum computers. Another application of LWE is in homomorphic encryption, serving as a hardness assumption for many important (fully) homomorphic encryption schemes (see Chapter 12).

RLWE is the LWE problem applied in rings of polynomials defined over finite fields. The RLWE problem represents a basis for future cryptography because it is resistant to known quantum algorithms such as Shor's algorithm, therefore it will remain a hardness assumption in the quantum ecosystem.

An advantage of the RLWE technique over LWE is the size of the keys. The size of the LWE keys is approximately the square of the size of the RLWE for the same number of bits of security. For example, for 128 bits of security, the keys of a LWE cryptosystem require 49,000,000 bits, while the keys of a RLWE cryptosystem require 7,000 bits.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_13

The RLWE cryptographic algorithms can be divided into three categories, as follows:

- **RLWE Key Exchange (RLWE-KE)**: In 2011, Jintai Ding, at the University of Cincinnati, used the associativity of the matrix multiplication to propose a preliminary scheme for key exchange based on LWE and RLWE [10]. The study was published in 2012, after the idea was patented. Based on this work, in 2014 Chris Peikert proposed a key transport scheme [11].
- **RLWE Signature (RLWE-S)**: The identification protocol proposed by Feige, Fiat, and Shamir in [12] was the basis for the digital signature proposed in 2011 by Lyubashevsky. A further improvement of the digital signature [13] was proposed by GLP (Gunesyu, Lyubashevsky, and Popplemann) in [14].
- **RLWE Homomorphic Encryption (RLWE-HE)**: In Chapter 12, you saw that homomorphic encryption enables computations to be applied directly over encrypted data. Among the first fully homomorphic encryption schemes that use RLWE is [15] and it was proposed in 2011 by Brakersky and Vaikuntanathan.

In the next section, we provide a minimum mathematical background for LWE and RLWE.

Mathematical Background

Learning with Errors

In the quantum computers era (where we are currently, although it is an early stage), a large number of the current encryption systems with public keys will be easily broken, which leads to the natural necessity of creating cryptosystems based on hardness assumptions that are quantum-resistant. LWE has this capability. Basically, the difficulty of the LWE problem consists in computing the values that solve this equation:

$$b = as + e$$

In an equation of this form, a and b can form the public key, s can be the secret key, and e can be an error value (or noise).

In cryptography, the LWE problem can be used in different topics. For example, based on LWE, public-key encryption schemes can be constructed that are secure against chosen plaintext or chosen ciphertext attacks. Also, LWE can be a basis for oblivious transfer, fully homomorphic encryption, or identity-based encryption.

The above equality becomes $b = A \times s + e$ in the work [1] because it is applied on linear equations. Here, A becomes a matrix with two dimensions and, if s is a matrix with one dimension, then b, e are matrices with one dimension. Another possibility is that A and b are matrices with one dimension and s is a scalar value.

Below, a simple encryption scheme based on LWE is presented [4]. Note that in the example, $p \in Z$ represents a prime number.

- **Key generation**: The following elements are chosen randomly: the vector $s \in \mathbb{Z}_p^n$, the matrix A with m rows which are m independent vectors of a uniform distribution, and the vector $e = (e_1, \ldots, e_m)$ of an error distribution defined over Z. Then, the value b is computed $b = As + e$. The secret key is the value s and the public key is the pair (A, b).
- **Encryption**: Given the message $m \in \{0, 1\}$ that will be encrypted, choose randomly samples from A and b, achieving $v_A = \sum a_i$ and $v_b = \sum b_j - \frac{p}{2}m$. The values a_i and b_i represents the samples from A and b, respectively. The encryption of m is the pair (u, v).
- **Decryption**: Compute $val = v_b - sv_A \, (mod\ p)$. If $val \leq \frac{p}{2}$, then the message is $m = 0$; otherwise, the message is $m = 1$.

In the above example, you can see how LWE works. Examples of public key encryption schemes based on the LWE problem are [2] and the Lindner-Peikert encryption schemes.

LWE problems are divided into two categories: LWE search and LWE decision. Next, we present these two variants.

Definition (LWE Search): Let $m, n, p \in Z$ be integer values and let χ_s and χ_e be two distributions defined over the integer numbers set Z. Select the values $s \leftarrow \chi_s^n, e_i \leftarrow \chi_e$ and $a_i \leftarrow \mathcal{U}\left(\mathbb{Z}_p^n\right)$ and compute the value of $b_i := \langle a_i, s \rangle + e_i \, mod\ p$, where $i = 1, \ldots, m$. Given the tuple $(n, m, p, \chi_s, \chi_e)$, the *learning with errors search* variant problem consists of determining s knowing $\left(a_i, b_i\right)_{i=1}^m$.

In this definition, s represents a column-vector with n values, a_i represents a row-vector with n values from Z_p, and b represents a column-vector with m elements from Z_p. The representation $x \leftarrow S$ shows that x is a random variable selected from the finite set S.

Definition (LWE Decision): Let $n, p \in Z$ be integer values and let χ_s and χ_e be two distributions defined over the integer numbers set Z. Select the value $s \leftarrow \chi_s^n$ and pick two oracles as below:

- $\mathcal{O} : a \leftarrow \mathcal{U}\left(\mathbb{Z}_p^n\right), e \leftarrow \chi_e$; output $(a, \langle a, s\rangle + e \bmod p)$
- $U : a \leftarrow \mathcal{U}\left(\mathbb{Z}_p^n\right), u \leftarrow \mathcal{U}\left(\mathbb{Z}_p\right)$; output (a, u)

Given the tuple (n, p, χ_s, χ_e), the *learning with errors decision* variant means to differentiate between $\mathcal{O}$ and U.

Ring Learning With Errors

The LWE problem applied in rings of polynomials with coefficients in a finite field is called the Ring Learning with Errors problem. RLWE is used in different domains of cryptography, for example, in key exchange, homomorphic encryption and signatures. The functionalities of RLWE are similar to the functionalities of simple LWE. For RLWE, the variables a, b, s, e from the first equality are polynomials. Further, we show how the two definitions for LWE variants are adapted for RLWE.

Definition (RLWE Search): Let $n, p \in Z$ be integer values, with $n = 2^k$, let R be $R = \frac{\mathbb{Z}[X]}{X^n + 1}$ and $R_p = \frac{R}{pR}$, and let χ_s and χ_e be two distributions defined over the ring R_p. Select $s \leftarrow \chi_s$, $e \longleftarrow \chi_e$ and $a \leftarrow \mathcal{U}\left(R_p\right)$ and compute the value of $b \coloneqq as + e$. Given the tuple (n, p, χ_s, χ_e), the *ring learning with errors search* variant problem consists in determining s knowing (a, b).

In this definition, R_p is actually $R_p = \frac{\mathbb{Z}_p[X]}{X^n + 1}$.

Definition (RLWE Decision): Let $n, p \in Z_+$ be integer values and let χ_s and χ_e be two distributions defined over the ring R_p. Select the value $s \leftarrow \chi_s$ and pick two oracles as below:

- $\mathcal{O} : a \leftarrow \mathcal{U}\left(R_p\right), e \leftarrow \chi_e$; output $(a, as + e)$
- $U : a \leftarrow \mathcal{U}\left(R_p\right), u \leftarrow \mathcal{U}\left(R_p\right)$; output (a, u)

Given the tuple (n, p, χ_s, χ_e), the *ring-learning with errors decision* variant means to differentiate between $\mathcal{O}$ and U.

An encryption scheme based on the hardness assumption of RLWE is secure if the advantage of any algorithm $\mathcal{A}$ (called the attacker) with polynomial time in solving the RLWE problem is a negligible function.

Practical Implementation

LWE is quantum-resistant technique in cryptography. On the practical side of LWE, to implement a simple LWE example, we first need to generate a secret value and a random value. Further, the implementation is intuitive, as we need to compute a value of the form `p[]=t[]×sk + e`.

In Listing 13-1, we provide an implementation for a simple example of encryption system based on the work of Oded Regev from [4]. The result of running the program is provided in Figure 13-1.

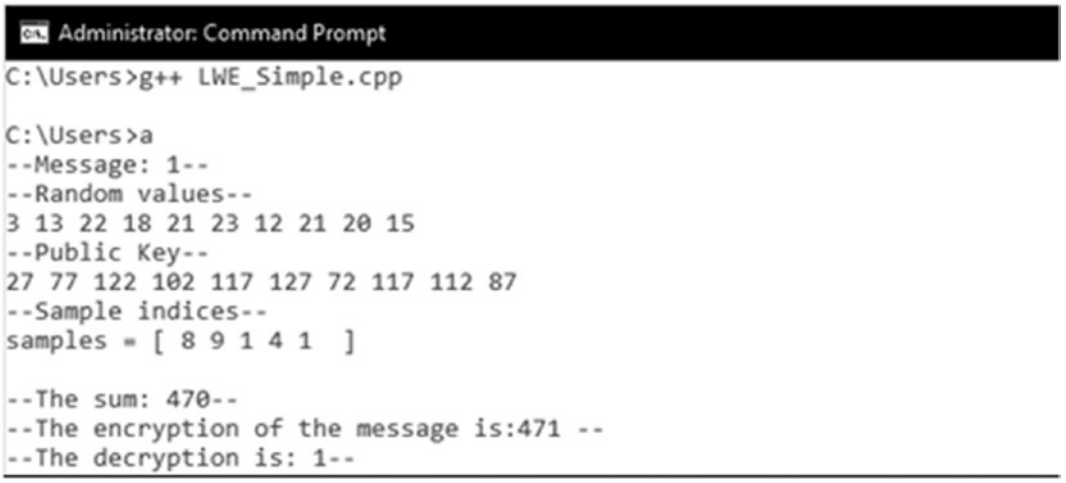


Figure 13-1. *The result of running the program with a simple example of LWE encryption*

Listing 13-1. Implementation of a Simple LWE Example Based on the Work [4]

```
#include <iostream>
#include <math.h>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));

    int no_of_values = 10;
    int public_key [no_of_values];
    int values [no_of_values];
    int secret_key = 5;
    int error_value = 12;
    int message = 1;
    int value = 0;

    for (int i = 0; i < no_of_values; i++)
    {
        //** generate random values between 0 and 23
        values[i] = rand() % (23 + 1 - 0) + 0;
        //** compute the public key
        public_key[i] = values[i] * secret_key + error_value;
    }

    cout<<"--Message: "<< message<<"--";
    cout<<endl<<"--Random values--"<<endl;
    for(int i = 0; i < no_of_values; i++)
    {
        cout<<values[i]<<" ";
    }

    cout<<endl<<"--Public Key--"<<endl;
    for(int i = 0; i < no_of_values; i++)
    {
        cout<<public_key[i]<<" ";
    }
```

```
    //** get half random samples from the public_key
    int noOfSamples = floor(no_of_values / 2);
    int samples [noOfSamples];
    for(int i=0; i < noOfSamples; i++)
    {
        //** generate a number of 5 random indices between 0 and 10
        samples[i] = rand() % ((no_of_values-1) + 1 - 0) + 0;
    }
    cout<<endl<<"--Sample indices--";
    cout<<endl<<"samples = [ ";
    for (int i=0; i < noOfSamples; i++)
    {
        cout << samples[i] << " ";
    }
    cout<<" ]" << endl;

    int sum = 0;
    for (int i = 0; i < noOfSamples; i++)
    {
        sum += public_key[samples[i]];
    }

    cout<<endl<<"--The sum: " << sum << "--";

    if (message == 1)
        sum+=1;

    cout<<endl<<"--The encryption of the message is:" << sum <<" --";

    int decryption = sum % secret_key;

    if (decryption % 2 == 0)
        cout<<endl<<"--The decryption is: 0--";
    else
        cout<<endl<<"--The decryption is: 1--";

    return 0;
}
```

Further, we provide in Listing 13-2 a more complex example of public-key encryption that uses LWE, based on the work [5]. The result of running the program is shown in Figure 13-2.

Listing 13-2. Implementation of the LWE Encryption Method Proposed by Oded Regev in [5]

```
#include <iostream>
#include <math.h>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));

    int numberOfRandVals = 20;
    int values_A [20]; //** values_A is a set of random numbers; represents
    the public key
    int secretValue = 5; //** represents the secret key
    int values_error [numberOfRandVals]; //** represents the error values
    int values_B [numberOfRandVals]; //** values_B is computed based on
    values_A, secretValue, values_error; represents the public key

    int q = 97; //** q is a prime number

    //** generate random values
    //** the number of random values is numberOfRandVals = 20
    //** the range is 0 - q=97
    for(int i=0; i < numberOfRandVals; i++)
    {
        //** to generate a random value in a range MIN - MAX,
        //** we proceed as folloes: val = rand() % (MAX + 1 - MIN) + MIN;

        //** generate random values between 0 - 97
        values_A[i] = rand() % (q + 1 - 0) + 0;
        //** generate small error values, between 1 - 4
        values_error[i] = rand() % (4 + 1 - 1) + 1;
```

```
    //** compute values_B using the formula B_i = A_i*s + e_i
    values_B[i] = values_A[i]*secretValue + values_error[i];
}

cout<<"--------- The parameters and the keys ---------" << endl;
cout<<"--Prime number (q)--" << endl;
cout<<"q = " << q << endl;
cout<<"--Public key (A, B)--" << endl;
cout<<"A = [ ";
for (int i=0; i < numberOfRandVals; i++)
{
    cout << values_A[i] << " ";
}
cout<<"]" << endl;

cout<<"B = [ ";
for (int i=0; i < numberOfRandVals; i++)
{
    cout << values_B[i] << " ";
}
cout<<"]" << endl;
cout<<"--Secret key (s)--" << endl;
cout<<"s = " << secretValue << endl;
cout<<"--Random error (e)--" << endl;
cout<<"e = [ ";
for (int i=0; i < numberOfRandVals; i++)
{
    cout << values_error[i] << " ";
}
cout<<"]" << endl;

cout<< endl << endl << "--------- Getting samples from the public
key... ---------";
int noOfSamples = floor(numberOfRandVals / 4); //** represents the
number of samples from the public key
int samples [noOfSamples];
```

```
for(int i=0; i < noOfSamples; i++)
{
    //** generate a number of 5 random indices between 0 and 19
    samples[i] = rand() % ((numberOfRandVals-1) + 1 - 0) + 0;
}
cout<<endl<<"--Sample indices--";
cout<<endl<<"samples = [ ";
for (int i=0; i < noOfSamples; i++)
{
    cout << samples[i] << " ";
}
cout<<" ]" << endl;
cout<<"--Sample pairs--";
for (int i=0; i < noOfSamples; i++)
{
    cout << endl <<"Sample " << i << ": ["
    << values_A[samples[i]] << " " << values_B[samples[i]] << "]";
}

cout<< endl << endl << "--------- Computing u and v... ---------";
int message = 0; //** the message to be encrypted can be a value from
{0, 1}
int u = 0, v = 0;
//** u = (sum (samples from values_A)) mod q
//** v = (sum (samples from values_B) + [q/2] * message) mod q
for (int i=0; i < noOfSamples; i++)
{
    u = u + values_A[samples[i]];
    v = v + values_B[samples[i]];
}

v = v + floor(q/2) * message;

u = u % q;
v = v % q;

cout<<endl<<"u = "<<u;
cout<<endl<<"v = "<<v;
```

```
    cout<< endl << endl << "--------- Encrypting... ---------";

    cout<<endl<<"--Message--";
    cout<<endl<<"m = "<<message;
    cout<<endl<<"--Encryption f the message--";
    cout<<endl<<"Enc(m) = (" << u << ", " << v <<")";

    cout<< endl << endl << "--------- Decrypting... ---------";
    int result = (v - secretValue * u) % q;

    int decryption;
    if (result > q/2)
        decryption = 1;
    else
        decryption  = 0;
    cout<<endl<<"The message is: " << decryption;

    return 0;
}
```

Figure 13-2. *The result of running the program of the public-key LWE example*

Listing 13-2 provides the example of public key encryption based on LWE, which was proposed in the work [5]. We first create a secret value `secretValue` which represents the private key. In the next step, we create the public key. The public key is formed by the values from a set of random numbers `values_A` and a set of values `values_B`, which are computed based on `values_A`, `secretValue`, and random errors `values_error`. This example is implemented for a single bit.

A simple workflow for this example is

- Between 0 and q (in the example, q=97), we select a random set of 20 values `values_A` that represent one of the components of the public key.
- Further, we define the set `values_B` where every element is computed as `values_B[i]=values_A[i] x secretValue+values_error[i] mod q`, where `secretValue` is the secret key, and where `values_error` represents a list of small random values called *the errors values.*
- The sets `values_A, values_B` form the public key and `secretValue` represents the secret key. At this point, we can share `values_A` and `values_B` with anyone who wants to proceed with an encryption of a message (with the condition to keep `secretValue` secret). In the encryption process, we use samples from `values_A` and `values_B.` Moving forward, based on those sample we take a bit message and compute the following two values:
  - $u = \sum (values\_A_{samples})(mod\ q)$
  - $v = \sum (values\_B_{samples}) + \frac{q}{2} \times message (\text{mod}\ q)$
- At this point, we can say that the encrypted message is (u, v). To proceed with the decryption, we need to compute
  - $decryption = v - s \times u\ (mod\ q)$
- If $decryption < \frac{q}{2}$, the message is equal with 0; otherwise 1.

The procedure described above is summarized from Oded Regev's paper [5] in order to make it easy to follow and to give you a clear understanding of how you can transpose the complexity of LWE in reality.

Conclusion

In this chapter, we discussed Ring Learning with Errors cryptography and we implemented two examples of encryption schemes using the C++ programming language as proposed in the works [4] and [5]. RLWE can be a space for many challenges for professionals and a starting point for significant contributions to this cryptographic primitive.

Through the chapter, you experienced an interesting journey with LWE from which you gained the following:

- A solid but short mathematical background of the main concepts and definitions on which RLWE is based and without which a practical implementation will have many gaps to fill
- Experience experimenting with the challenges brought by RLWE's mathematical concepts and their transposition in practice
- The ability to implement simple examples of public-key encryption schemes based on LWE

References

[1] Oded Regev, The Learning with Errors Problem, https://cims.nyu.edu/~regev/papers/lwesurvey.pdf.

[2] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography" in *Journal of the ACM (JACM)*, 56(6) (pp. 1-40). 2009.

[3] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption" in *Cryptographers' Track at the RSA Conference* (pp. 319-339). Springer; Berlin, Heidelberg. February, 2011.

[4] O. Regev, "The Learning with Errors Problem (Invited Survey)" in *2010 IEEE 25th Annual Conference on Computational Complexity* (pp. 191-204). Cambridge, MA. doi: 10.1109/CCC.2010.26. 2010.

[5] D. Micciancio and O. Regev. “Lattice-based cryptography” in (eds. D. J. Bernstein and J. Buchmann) *Post-quantum Cryptography*. Springer. 2008.

[6] C. Peikert, ”Some recent progress in lattice-based cryptography” in Slides for invited tutorial at TCC’09, 2009.

[7] D. Micciancio, “Cryptographic functions from worst-case complexity assumptions” in (eds. P. Q. Nguyen and B. Vall ́ee*) The LLL Algorithm: Survey and Applications, Infor-mation Security and Cryptography* (pp. 427–452). Springer. 2008. Prelim. version in LLL25, 2007.

[8] O. Regev, “Lattice-based cryptography” in *CRYPTO* (pp. 131–141). 2006.

[9] NewHope – Post-quantum Key Encapsulation. Available online: https://newhopecrypto.org/.

[10] *Jintai Ding, Xiang Xie, and Xiaodong Lin, “A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem.” January 1, 2012. Available online:* https://eprint.iacr.org/2012/688.

[11] C. Peikert, “Lattice Cryptography for the Internet” in (ed. M. Mosca) *Post-Quantum Cryptography. PQCrypto 2014. Lecture Notes in Computer Science, vol 8772*. Springer, Cham. 2014.

[12] Y. Desmedt, “Fiat–Shamir Identification Protocol and the Feige–Fiat–Shamir Signature Scheme” in (eds. H.C.A. van Tilborg and S. Jajodia) *Encyclopedia of Cryptography and Security*. Springer, Boston, MA. 2011.

[13] V. Lyubashevsky “Lattice Signatures without Trapdoors” in (eds. D. Pointcheval and T. Johansson) *Advances in Cryptology – EUROCRYPT 2012. EUROCRYPT 2012. Lecture Notes in Computer Science, vol 7237*. Springer; Berlin, Heidelberg. 2012.

[14] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann, "Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems" in (eds. Emmanuel Prouff and Patrick Schaumont) *Lecture Notes in Computer Science* (pp. 530–547). Springer; Berlin, Heidelberg. doi:10.1007/978-3-642-33027-8_31. ISBN 978-3-642-33026-1. 2012.

[15] Zvika Brakerski and Vinod Vaikuntanathan, "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages" in (ed. Phillip Rogaway) *Lecture Notes in Computer Science* (pp. 505–524). Springer; Berlin Heidelberg. doi:10.1007/978-3-642-22792-9_29. ISBN 978-3-642-22791-2. 2011.

CHAPTER 14

Chaos-Based Cryptography

In chaos-based cryptography, the chaos theory and its mathematical background are applied for creating novel and unique cryptographic algorithms. The first attempt of using the chaos theory in cryptography was initiated by Robert Matthews in 1989 through the work [1], which attracted much interest.

In contrast to the regular cryptographic primitives used daily, the chaos theory and its system are used in an efficient way by implementing the chaotic maps towards confusion and diffusion. Through this chapter, the cryptographic algorithm is referred to as the chaotic system.

To understand the similarities and differences between chaotic systems and cryptographic algorithms, we present a set of correspondences in Table 14-1 introduced by L. Kocarev in [2].

Table 14-1. *Similarities and Differences Between Chaotic Systems and Cryptographic Algorithms*

Chaotic System	Cryptographic Algorithm
Phase space: (sub) set of real numbers	Phase space: finite set of integers
Iterations	Rounds
Parameters	Key
Sensitivity to a change in initial conditions and parameters	Diffusion
?	Security and Performance

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_14

Further, we demonstrate the similarities and the differences from Table 14-1 using an example of a chaotic system, the shift map:

$$x(t+1)=ax(t)(mod\,1)$$

where the phase space x=[0,1] is the unit interval and a>1 is an integer value.

From the chaos theory perspective, cryptography can use different functions and discrete-time systems. By analyzing them, the phase space will become a finite set of integers and the parameters will be inter values. The version of the shift map that uses the discrete phase-space is one of the common examples:

$$p(t+1)=ap(t)(mod\,N)$$

where $a > 1$, N and p are integer values, with the restrictions $p \in [0, 1, \ldots, N-1]$, and N is coprime to a. This representation of the shift map is invertible, which means that all trajectories placed within a dynamical system with a finite phase space are called *periodical*. This fact introduces a new concept, namely the period function P_N that describes the least period of the map F, denoted F^{P_N} as its identity, and P_N is minimal as it is a function within a system of size N.

Another very important metric used in the practical chaotic systems is the *Lyapunov exponent* (LE), whose trivial value is 0. The reason for it is the case in which the orbit is periodic and it will reiterate itself.

With this information, below are presented two concepts of block diagrams (for text encryption and image encryption) that demonstrate an encryption scheme based on the chaos theory. Figures 14-1 and 14-2 show the encryption process and the decryption process, respectively, based on the logistic map. Figure 14-3 shows an example of image encryption and decryption.

Following the examples of the block diagrams, we can examine the former papers and the other papers listed in this chapter's references to observe that the encryption models and the way in which they are built are different according to the chaotic map used. Before designing new cryptographic approaches and mechanisms based on the chaos theory, it is very important to comprehend the way in which different chaotic maps work.

A good starting point is to use the following block diagrams as a guide from theory to practice, because the models are created according to the similarities and differences from Table 14-1.

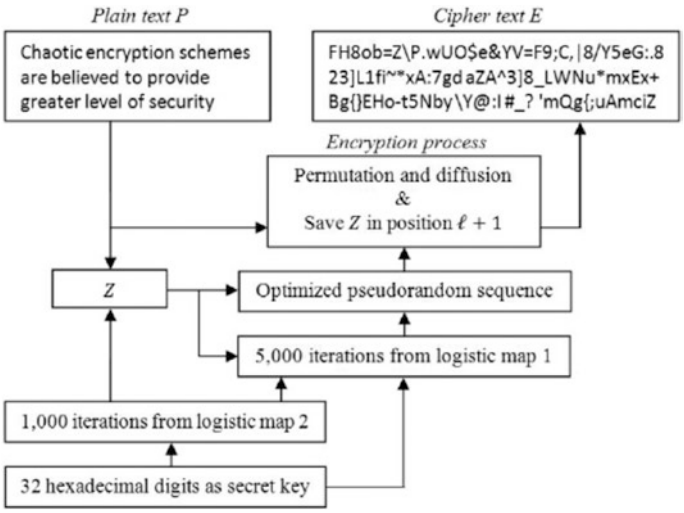


Figure 14-1. *Block diagram for text encryption using logistic map [14]*

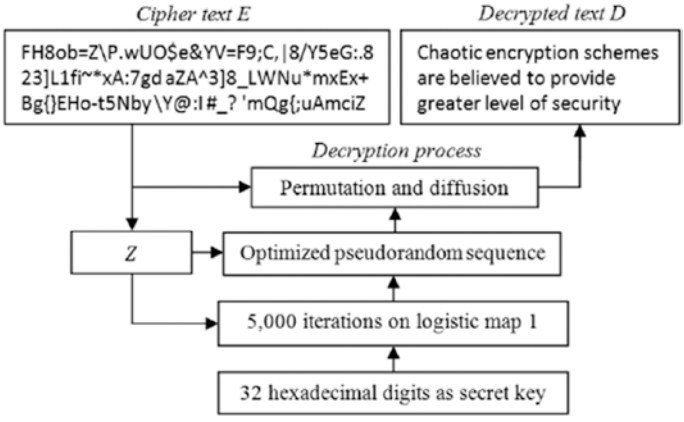


Figure 14-2. *Block diagram for text decryption using logistic map [14]*

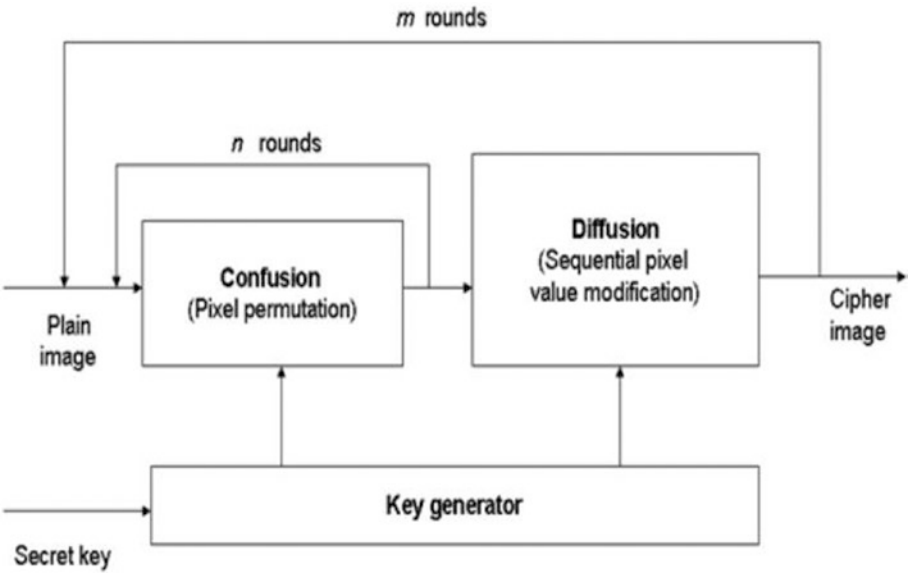


Figure 14-3. *Block diagram for an image encryption cryptosystem [15]*

Security Analysis

In this section, we present a security analysis in the form of techniques for finding the weakness or security breaches in the cryptosystem and then we get a piece or the whole encrypted image or plaintext or find the key without knowing the algorithm or the decryption key.

Examples of attacks over encrypted images are presented in [3] and [4]. The following methods, techniques, and analysis should be considered in designing a chaotic system or in conducting a cryptanalytic attack:

- **Key space analysis**: This is the number of trials for finding the decryption key, and it is made by trying all possible keys from the keyspace of the encryption system. An important remark is that the keyspace grows exponentially at the same time with the incrementation of the key's size.
- **Key sensitivity analysis**: For a good encryption system for images, an important thing that should be considered is the sensitivity of the secret key. If just a single bit is modified in the secret key, then the output image should be a completely different image (regarding encryption or decryption).

- **Statistical analysis**: The purpose of this analysis is to prove the relationship between the original image and the encrypted one.
- **Correlation coefficient analysis**: An important graphical tool that needs to be studied is the histogram, namely the distribution of the values generated by a trajectory of a dynamic system. Among the histogram analysis, the correlation between the pixels of a plain image and the encrypted image is another important technique, as it is made between two pixels distributed vertically, horizontally, and diagonally.
- **Information entropy analysis**: The analysis based on the entropy tests the robustness of the encryption algorithm. The comparison between the entropy of the plain image and the encrypted image is very important, which shows that the entropy of the encrypted images is about an 8-bit depth. This is useful in proving the encryption technique against the entropy attack.
- **Differential analysis**: The differential analysis determines the sensitivity of the cryptosystem regarding any slight change in the algorithm. The sensitivity can be computed based on two criteria: NPCR (number of pixels change rate) and UACI (unified average changing intensity). When these two test are made, the high values show the small changes that occurred in the plain image, which produced significant modifications in the encrypted image.

Chaotic Maps for Plaintexts and Images Encryption

This section presents chaotic maps with respect to their encryption target (text encryption or image encryption).

Many encryption algorithms for images from the below list (Table 14-2) have been analyzed and tested by the authors who proposed them by using the techniques described above. It is useful to have validation of the performance and to have an evaluation of the robustness of the encryption scheme. All the references were analyzed and chosen as good references based on their analysis and tests.

Table 14-2. *Chaotic Map (Systems) for Image Encryption*

Chaotic Map (System)	Metrics Entropy	NPCR	UACI	Key Space	Sensitivity	References
Lorenz Baker	7.9973	-	-	2^{128}	High	[5]
Lorenz	-	-	-	Large	Medium	[6]
Henon Map	7.9904	0.0015%	0.0005%	2^{128}	High	[7]
Logistic Map	7.9996	99.6231%	33.4070%	10^{45}	High	[8]
Trigonometry Maps	-	0.25%	0.19%	2^{302}	-	[9]
Arnold Cat Map	7.9981	99.62%	33.19%	2^{148}	High	[10]
Chebyshev Map	7.9902	99.609%	33.464%	2^{167}	High	[11]
Circle Map	7.9902	99.63%	33%	2^{256}	High	[12]
Arnold Map	-	0.0015%	0.004%	-	-	[13]

Rössler Attractor

The Rössler attractor is a system that is formed from three non-linear ordinary differential equations. The equations define a continuous-time dynamical system that exposes chaotic dynamics, which are associated with the fractal properties of the attractor.

The equations of the Rössler system are as follows:

$$\begin{cases} \frac{dx}{dt} = -y - z \\ \frac{dy}{dt} = x + ay \\ \frac{dz}{dt} = b + z(x - c) \end{cases}$$

When Rössler is applied in real life, computing the fixed points is one of the first challenges. To compute the fixed points, it is sufficient that the equations are set to

zero and the (x, y, z) coordinates of each of the fixed points are computed by solving the resulting equations. We have the following general equations of each of the fixed point coordinates:

$$\begin{cases} x = \dfrac{c \pm \sqrt{c^2 - 4ab}}{2} \\ y = -\left(\dfrac{c \pm \sqrt{c^2 - 4ab}}{2a} \right) \\ z = \dfrac{c \pm \sqrt{c^2 - 4ab}}{2a} \end{cases}$$

These equations are turned in such way that will show the current fixed points that are given for a set of values associated with the parameters.

$$\left(\frac{c + \sqrt{c^2 - 4ab}}{2}, \frac{-c - \sqrt{c^2 - 4ab}}{2a}, \frac{c + \sqrt{c^2 - 4ab}}{2a} \right)$$

$$\left(\frac{c - \sqrt{c^2 - 4ab}}{2}, \frac{-c + \sqrt{c^2 - 4ab}}{2a}, \frac{c - \sqrt{c^2 - 4ab}}{2a} \right)$$

The above equations are used in our example from Listing X, where we implement a solution for generating secure random numbers using the chaos perspective of a Rössler attractor.

Complex Numbers – Short Overview

Complex numbers represent an extension of real numbers. The motivation behind complex numbers is in the desire to provide a way of solving algebraic equations that normally (using traditional real numbers) have no solution. As an example, $x^2 + 1 = 0$ has no real solution. For this situation, a symbolic solution has been created and it is known as the *imaginary unit i,* which has the following property:

$$i^2 = -1$$

A complex number is represented with two components, which are known as *the real part* and *the imaginary part*. We have the following:

$$z = x + yi$$

where $real(z) = x$ denotes the real part, $imag(z) = y$ the imaginary part, and i represents the imaginary unit.

The arithmetic behind the complex numbers is quite straightforward and it is an extension of the arithmetic of real numbers. To understand the previous statement, we define two numbers z and w as follows:

$$z + w = (x + yi) + (u + vi) = (x + u) + (y + v)i.$$

We add the real and imaginary components separately. The next step is to multiply the numbers as follows:

$$z \cdot w = (x + yi)(u + vi) = xu + xvi + yui + yvi^2 = (xu - yv) + (xv + yu)i.$$

Observe that yvi^2 represents the real part of the product becoming $-yv$, due the property defined above, namely $i^2 = -1$.

In the example presented in Listing X we use complex numbers with chaos and fractals properties to provide encryption and decryption operations.

Practical Implementation

The applications and programs that use chaotic systems have applicability for *plaintext encryption* and *image encryption*. If we look at other areas of cryptography (such as the ones discussed in this book), the research community has a significant amount of theoretical contributions. The lack of practical implementations and directions has raised multiple difficulties and challenges for researchers and professionals.

If we look at the practicality of chaos cryptography, there are not many practical implementations. Below we will list some of the practical approaches (and here we are referring to pseudocode algorithms) that can be found within [16]. The work from [16] provides a very in-depth structure and very good ideas and approaches on how different cryptosystems based on chaos theory can be implemented. The ideas are provided as pseudocode. The work covers the following cryptosystem types:

- Chaos-based public-key cryptography
- Pseudo-random number generation in cryptography
- Formation of high-dimensional chaotic maps and their uses in cryptography
- Chaos-based hash functions
- Chaos-based video encryption algorithms
- Cryptanalysis of chaotic ciphers
- Hardware implementation of chaos-based ciphers
- Hardware implementation of chaos-secured optical communication systems

In [16], starting with Chapter 2, the authors propose an interesting public-key cryptosystem that uses the chaos approach and consists of three steps: a key generation algorithm (see Pseudocode 14-1), an encryption algorithm (see Pseudocode 14-2), and a decryption algorithm (see Pseudocode 14-3). The scenario is a typical communication between two user entities, Alice and Bob. Below we will provide the structure of each algorithm and at the end we will provide implementations for demonstrating the applicability.

Pseudocode 14-1. *Key Generation Algorithm [16]*

`Start. Alice needs to generate the keys before the communication. For this she will accomplish the following:`

- A large integer a has to be generated.
- Calculate $G_a(p)$ based on a random number selected as $p \in [-1, 1]$.
- Alice will set her public key as $(p, G(p))$ and her private key to a.

Pseudocode 14-2. *Encryption Algorithm [16]*

`Start.` *`Bob wants to encrypt a messsage. To achieve this, the following must be done`*`:`

- Get Alice's authentic public key $(p, G_a(p))$.
- Calculate and represents the message as a number $M \in [-1, 1]$.

- Generate a large integer r.
- Calculate $G_r(p)$, $G_{r\cdot a(x)} = G_r(G_a(p))$ and $X = M \cdot G_{r\cdot s}(p)$.
- Take the ciphertext and send it as $C = (G_r(p), X)$ to Alice.

Pseudocode 14-3. *Decryption Algorithm [16]*

`Start.` *`Alice wants to read the text and to do this she will have to recover M from the ciphertext C. To achieve this, the following steps are done:`*

- Alice has to use her private key a and to calculate $G_{a\cdot t} = G_a(G_r(p))$.
- The message M will be obtained by calculating $M = \frac{X}{G_{a\cdot r}(p)}$.

Secure Random Number Generator Using a Chaos Rössler Attractor

In this section, we will present the implementation of a secure random number generator using a chaos Rössler attractor. The application has five files (`encryption.h`, `generation.h`, `encryption.c`, `generation.c`, and `chaos_random.cpp`). To compile and run the application, we need to run the following command in the terminal:

```
g++ -o test.exe chaos_random.cpp generation.c generation.h encryption.c
encryption.h
```

Below we will examine each of the files and we will discuss the most important lines of code.

Figure 14-4 shows the execution of the program and the numbers generated for each of the keys. As you saw in the Rössler attractor section, there are three fixed points that need to be computed in order to solve the equations. Each fixed point is represented by a cryptographic key (e.g. *key 1, key 2, key 3).*

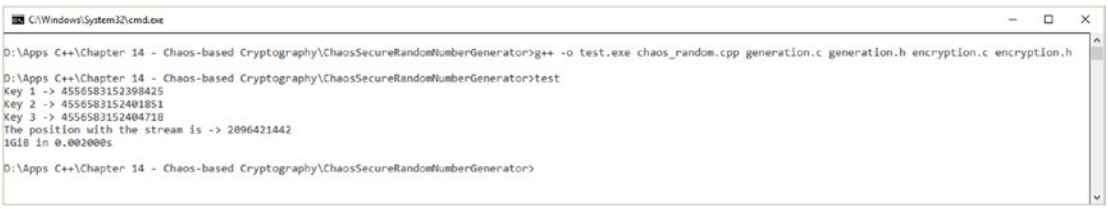


Figure 14-4. *Secure random number generator*

Listing 14-1 contains the header file (`encryption.h`) for defining the signature function for encryption process, `encryption`. The function has three input values:

- `struct generation *g`: A struct object used to generate the mantisa, exponent, and sign for obtaining the normalization form of a real number. The definition of the struct can be found within the file `generation.h` (see Listing 16-2).
- `uint8_t *buffer`: The buffer with the data used for encryption
- `size_t length`: The length of the buffer

Listing 14-1. Header File encryption.h

```
#ifndef ENCRYPTION_H
#define ENCRYPTION_H

#include "generation.h"
#include <stddef.h>

void encryption(struct generation *g, uint8_t *buffer, size_t length);

#endif
```

Listing 14-2 contains the implementation of the `generation.h` header file, which contains definitions for the Rössler attractor (see `ROSSLER(x,n)`), the coordinates (`A, B,` and `C`), integral approximation (`APPROXIMATION constant`), removing noise constant (`REMOVE_NOISE`), two functions for generating the initialization on 16 and 32 bits (`generation_initialization` and `generation 32`), describing the normalization of real numbers as a union and struct types, containing for the double numbers the mantisa, exponent, and sign (`realbits` union), and a struct (`generation` struct) for the generation process that contains three variables which represent the fixed points (e.g. `x, y,` and `z`).

Listing 14-2. Header File generation.h

```
#ifndef GENERATION_H
#define GENERATION_H

#include <inttypes.h>
#include <math.h>
```

```
// the Rossler (ROL) attractor definition for plane (x,n)
#define ROSSLER(x,n) ((x = ((x << n) | (x >> (32 - n)))))

// the attractor variables (coordinates) - for this example Rossler is chosen
#define A_Coordinate 0.5273
#define B_Coordinate 3
#define C_Coordinate 6

// constant for integral approximation as a step size
#define APPROXIMATION 0.01

// constant used for removing the initial noise
#define REMOVE_NOISE 64

void generation_initialization(struct generation *g, uint64_t k[3]);
uint32_t generation32(struct generation *g);

// the normalization form of a real number
union realbits
{
      double d;
      struct
      {
                  uint64_t mantisa: 52;
                  uint64_t exponent: 11;
                  uint64_t sign: 1;
      } rb;
};

struct generation
{
      union realbits x, y, z;
};

#endif
```

Listing 14-3 contains the implementation function for the encryption process. Note the fact that the `encryption.c` source file includes both header files from Listing 14-1 and Listing 14-2. As mentioned, the encryption is done using a generation struct that contains three fixed points, a buffer used to hold the content being encrypted, and its

length. The function is quite self-explanatory and the main idea behind it is based on the position within the data stream, and the number of calls plays an important role as it is using the length of the buffer and shifting to the right 2 bits.

Listing 14-3. File encryption.c

```
#include "encryption.h"
#include "generation.h"

#include <iostream>

using namespace std;

// performing the encryption operation
void encryption(struct generation *g, uint8_t *buffer, size_t length)
{
        uint32_t position_in_stream;
        size_t number_of_calls = length >> 2;
        size_t l_neighbour = length & 3;
        uint8_t *temporary = (uint8_t *)&position_in_stream;

        for(size_t index = 0; index < number_of_calls; ++index)
           {
                 position_in_stream = generation32(g);
                 buffer[(index<<2)] ^= temporary[0];
                 buffer[(index<<2)+1] ^= temporary[1];
                 buffer[(index<<2)+2] ^= temporary[2];
                 buffer[(index<<2)+3] ^= temporary[3];
        }

        if(l_neighbour != 0)
      {
                  position_in_stream = generation32(g);
                  for(size_t index = 0; index < l_neighbour; ++index)
                     buffer[(number_of_calls<<2)+index] ^= temporary[index];
        }

         std::cout<<"The position with the stream is -> "<<position_in_
         stream<<endl;
}
```

Listing 14-4 contains the implementation for the different operations necessary for generating the fixed points and performing the initialization process. Here we also use the ROSSLER function defined in Listing 14-2.

Listing 14-4. File generation.c

```
#include "generation.h"

static void initialization(struct generation *gen, double initValueX,
double initValueY, double initValueZ)
{
      gen->x.d = initValueX;
      gen->y.d = initValueY;
      gen->z.d = initValueZ;
}

static void perform_iteration(struct generation *gen)
{
      gen->x.d = gen->x.d + APPROXIMATION * (-gen->y.d - gen->z.d);
      gen->y.d = gen->y.d + APPROXIMATION * (gen->x.d + A_Coordinate *
      gen->y.d);
      gen->z.d = gen->z.d + APPROXIMATION * (B_Coordinate + gen->z.d *
      (gen->x.d - C_Coordinate));
}

void generation_initialization(struct generation *gen, uint64_t keyValue[3])
{
      initialization(gen,
                (double)keyValue[0] / 9007199254740992,
                (double)keyValue[1] / 8674747684896687,
                (double)keyValue[2] / 6758675765879568);

     for(uint8_t index = 0; index < REMOVE_NOISE - 1; ++index)
            perform_iteration(gen);
}
```

```
uint32_t generation32(struct generation *gen)
{
	uint32_t message[6];
	message[0] = (uint32_t)(gen->x.rb.mantisa >> 32);
	message[1] = (uint32_t)(gen->x.rb.mantisa);
	message[2] = (uint32_t)(gen->y.rb.mantisa >> 32);
	message[3] = (uint32_t)(gen->y.rb.mantisa);
	message[4] = (uint32_t)(gen->z.rb.mantisa >> 32);
	message[5] = (uint32_t)(gen->z.rb.mantisa);
	perform_iteration(gen);

	message[0] += message[1];
	message[2] += message[3];
	message[4] += message[5];

	for(uint8_t index = 0; index < 4; ++index)
	{
		ROSSLER(message[0],7); ROSSLER(message[3],13);
		message[5] ^= (message[4] + message[3]);
		message[1] ^= (message[2] + message[0]);
		message[2] = message[2] ^ message[0] ^ message[5];
		message[4] = message[4] ^ message[3] ^ message[1];
	}

	message[2] += message[4];
	return message[2];
}
```

Listing 14-5 contains the implementation of the main program. Note that the path to the file which contains random numbers (similar to `urandom` from the UNIX OS) has to be adjusted accordingly to reader comfort. The line where the path must be modified is shown with bold as follows:

```
if((folder = open("D:/Apps C++/Chapter 14 - Chaos-based Cryptography/
ChaosSecureRandomNumberGenerator/dev/urandom", O_RDONLY)) == -1)
```

Listing 14-5. Main Program

```
#include "encryption.h"
#include "generation.h"

#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <windows.h>
#include <time.h>
#include <inttypes.h>
#include <iostream>

using namespace std;

const size_t MESSAGE_LENGTH = 2000000000;

uint64_t generateStringOfBytes()
{
        int folder = 0;
        ssize_t resourceFile = 0;
        uint64_t buffer = 0;

        if((folder = open("D:/Apps C++/Chapter 14 - Chaos-based
        Cryptography/ChaosSecureRandomNumberGenerator/dev/urandom",
        O_RDONLY)) == -1)
         exit(-1);

         if((resourceFile = read(folder, &buffer, sizeof buffer)) < 0)
            exit(-1);

        buffer &= ((1ULL << 53) - 1);
        close(folder);
        return buffer;
}
```

```
int main(void)
{
        struct generation gen;
        uint64_t key[3] = {generateStringOfBytes()+rand()%3000, generate
        StringOfBytes()+rand()%5000, generateStringOfBytes()+rand()%8000};
            cout<<"Key 1 -> "<<key[0]<<endl;
            cout<<"Key 2 -> "<<key[1]<<endl;
            cout<<"Key 3 -> "<<key[2]<<endl;

        // generate 1GiB of 1s
        uint8_t *message = (uint8_t*)malloc(MESSAGE_LENGTH);
        memset(message, 1, MESSAGE_LENGTH);

        // perform encryption
        generation_initialization(&gen, key);
        clock_t s = clock();
        encryption(&gen, message, MESSAGE_LENGTH);
        clock_t e = clock();
        double spent = (double)(e - s) / CLOCKS_PER_SEC;
        printf("1GiB in %lfs\n", spent);

        free(message);
}
```

Cipher Using Chaos and Fractals

In this section, we will discuss and implement a solution for the encryption/decryption operation using chaos and fractals notions.

Listing 14-6 contains the declaration of the main functions that deal with processing the representation of the starting points and performing the projections for both axes, `x` and `y`. It is necessary to mention that one of the most challenging operations and tasks when using fractals and chaotic systems is to identify the path and to identify the main root (see function `identifyFirstRoot()`). The code in Listings 14-6 and 14-7 is quite self-explanatory and contains the necessary notes to be fully understandable. See Figure 14-5 for the execution of the application.

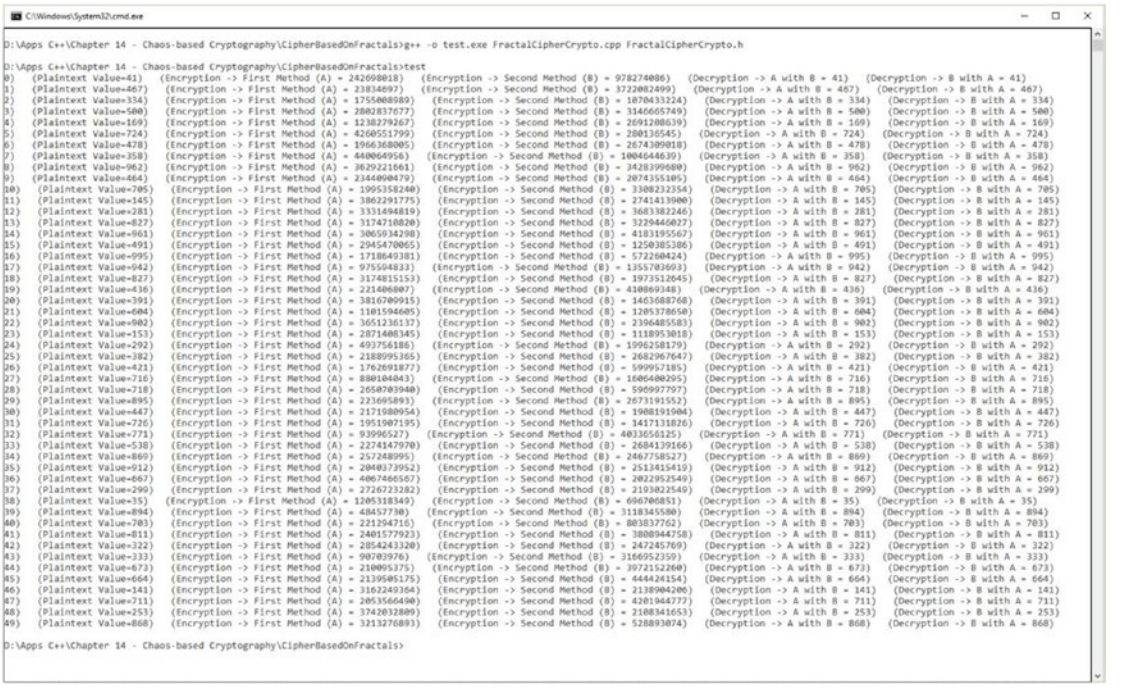


Figure 14-5. *Execution of the encryption/decryption process*

To run the program, the following command must be entered in the terminal:

```
g++ -o test.exe FractalCipherCrypto.cpp FractalCipherCrypto.h
```

Listing 14-6. Header File FractalCipherCrypto.h

```
#ifndef CRYPTOCIPHERFRACTALS_H_
#define CRYPTOCIPHERFRACTALS_H_

#include <climits>
#include <assert.h>
#include <math.h>

class CryptoFractalCipher
{
      // point C = (x, y) - the representation in the xOy system of point C
      double c_xCoordinatePoint, c_yCoordinatePoint;

      // point Z = (x,y) - the representation in the xOy system of point Z
      double z_xCoordinatePoint, z_yCoordinatePoint; //Zx,Zy;
```

```
  // get the sign of a double number
  inline double getSign(double number)
  {
      // in case that d is less than 0, return -1.0, making the number
         negative
      // contrary make the number positive
     if (number<0)
        return(-1.0);
     else
        return(1.0);
 };

 // Value 'yValue' will be projected over an integer matrix or grid.
 // We have choose this for achieving the scaling goal and performing
    tests.
 // The projection process is a matter of personal choice, any other
    idea or
 // solution can be implemented by reader.
 inline unsigned int PerformProjectionFor_Y(double yValue)
{
    unsigned long q;
    const double scale=(32768.0/2.0);
    const double offset=(32768.0);

    // do the projection as a positive integerproject to positive
       integer
  q=(yValue*scale)+offset;

   //getting the LSB (least significant bit)
  q&=1;
  return q;
}

  // Value 'xValue' will be projected over an integer matrix or grid.
  // We have choose this for achieving the scaling goal and performing
     tests.
```

```
    // The projection process is a matter of personal choice, any other
       idea or
    // solution can be implemented by reader.
    inline unsigned int PerformProjectionFor_X(double xValue)
{
        // used for storing the decomposition value
        double decompositionValue;

        // power value (exponent)
      int n;

        // with frexp() we will decompose the double point (xValue) as
       // argument into a normalized fraction and an integral power
      decompositionValue = frexp (xValue , &n);

       // with ldexp() we will return the result of multiplying
          'decompositionValue'
       // (the significand) with 2 and raised to the power '51'
          (exponent)
      decompositionValue = ldexp(decompositionValue,51);

       // Test if the difference between 'decompositionValue' and
       // floor(decompositionValue) is less than 0.5
       // if yes return '1', otherwise '0'.
       // With floor() we round 'decompositionValue', returning the
          largest
       // integral value that is not greater than 'decompositionValue'
      return (((decompositionValue-floor(decompositionValue))<0.5)?1:0);
}

 inline void identifyFirstRoot()
 {
     /* Zn*Zn=Z(n+1)-c */
      z_xCoordinatePoint=z_xCoordinatePoint-c_xCoordinatePoint;
      z_yCoordinatePoint=z_yCoordinatePoint-c_yCoordinatePoint;
```

```
// r represents the length of the vector from the origin to the
   point
// r = |z| = sqrt(x*x+y*y)
double r;

// the new point z = (x,y)
double z_xNewPointValue, z_yNewPointValue;          //NewZx, NewZy
r=sqrt(z_xCoordinatePoint*z_xCoordinatePoint+z_
yCoordinatePoint*z_yCoordinatePoint);

// the below code sequence represents the implementation of the
   algorithm presented in [17], from page 361 to 362.
// case 1: z>0
if (z_xCoordinatePoint>0)
{
     z_xNewPointValue=sqrt(0.5*(z_xCoordinatePoint+r));
     z_yNewPointValue=z_yCoordinatePoint/(2*z_xNewPointValue);
}

// for cases when z<0 and z=0
else
{
     // case 2: z<0
     if (z_xCoordinatePoint<0)
     {
          z_yNewPointValue=getSign(z_yCoordinatePoint)*sqrt(0.5*(
          -z_xCoordinatePoint+r));
          z_xNewPointValue=z_yCoordinatePoint/(2*z_yNewPointValue);
     }

     //case 3: z=0
     else
     {
          z_xNewPointValue=sqrt(0.5*fabs(z_yCoordinatePoint));
          if (z_xNewPointValue>0) z_yNewPointValue=z_
          yCoordinatePoint/(2*z_xNewPointValue);
          else z_yNewPointValue=0;
     }
```

```
          };
          // end of the implementation

          // the values for x and y coordinates
          z_xCoordinatePoint=z_xNewPointValue;
          z_yCoordinatePoint=z_yNewPointValue;
     };

public:
     // gets the encrypted value
     unsigned int getEncryptedMessageA(unsigned int plainValue);
     unsigned int getDecryptedMessageB(unsigned int encryptedValue);
     unsigned int getEncryptedMessageC(unsigned int stream);
     unsigned int getDecryptedMessageD(unsigned int stream);

     // gets the single bit
     unsigned int bitCodeEncryptedMessageA(unsigned int plainValue);
     unsigned int bitCodeDecryptedMessageB(unsigned int encryptedValue);
     unsigned int bitCodeEncryptedMessageC(unsigned int stream);
    unsigned int bitCodeDecryptedMessageD(unsigned int stream);

     // constructor
     CryptoFractalCipher(double cx,double cy);

     // destructor
     virtual ~CryptoFractalCipher();
};
#endif
```

Listing 14-7. Main Program

```
#include "FractalCipherCrypto.h"
#include <climits>
#include <assert.h>
#include <math.h>
#include <iostream>

using namespace std;
```

```
// implementing bitCodeEncryptedMessageA from FractalCipherCrypto.h file
unsigned int CryptoFractalCipher::bitCodeEncryptedMessageA(unsigned int
bit_from_plaintext)
{
     // below we will create a cryptographic strem from the clear stream
    int crypto_bit=0;
    {
        identifyFirstRoot();

          // quadratic value
        unsigned long quadraticValue = PerformProjectionFor_X(
        z_yCoordinatePoint);

          // Do the encoding process and provide the
          // cryptographic stream from the clear stream
          // Variables used:
          //          - iV: the input value
          //          - oV: the output value
          //          - rV: the route value in the expansion of the fractal
        unsigned int iV, oV, rV;
        {
               unsigned int result1, result2, result3;
            iV=(bit_from_plaintext) & 1;

               // obtained from the iteration of the quadratic value
            result1=quadraticValue;

               // input value
            result2=iV;

               // we will copy the bits if it is set in one operand but not
                  both
            result3=result1^result2;

               // the final output value
            oV=result3;

               // the route value that need to be followed within the
                  expansion of the fractal
```

```
                rV=result2;
        }
        crypto_bit=(oV);

        if ((rV) != 0)
        {
            // use the route on the second root point
            z_xCoordinatePoint=-z_xCoordinatePoint;
            z_yCoordinatePoint=-z_yCoordinatePoint;
        }
    }
    return crypto_bit;
};

unsigned int CryptoFractalCipher::bitCodeDecryptedMessageB(unsigned int
bit_from_encoding)
{
     // decode the clear value from the cryptographic stream
    int bit_from_plaintext=0;
    {
        identifyFirstRoot();

          // computing the quadratic value
        unsigned long quadraticValue = PerformProjectionFor_X(z_
        yCoordinatePoint);

          // decoding process for obtaining the clearstream from the
             cryptographic stream
          // Variables used:
          //          - iV: the input value
          //          - oV: the output value
          //          - rV: the route value in the expansion of the fractal
        unsigned int iV, oV, rV;
        {
               unsigned int result1,result2,result3;

            iV=(bit_from_encoding) & 1;
```

```
            // obtained from the iteration of the quadratic value
        result1=quadraticValue & 1;

            // input value
        result3=iV;

            // we will copy the bits if it is set in one operand but not
               both
        result2=result1^result3;

            // the output value
        oV=result2;

            // the route value that need to be followed within the
               expansion of the fractal
        rV=result2;
    }
    bit_from_plaintext=(oV);

    if ((rV) != 0)
    {
            // use the route on the second root point
        z_xCoordinatePoint=-z_xCoordinatePoint;
        z_yCoordinatePoint=-z_yCoordinatePoint;
    }
  }
  return bit_from_plaintext;
};

unsigned int CryptoFractalCipher::bitCodeEncryptedMessageC(unsigned int
bit_from_stream)
{
   // generate the cryptographic stream from the clear stream
  int bit_from_coding=0;
  {
    identifyFirstRoot();
```

```
        unsigned long quadraticValueForY = PerformProjectionFor_X(
        z_yCoordinatePoint);
        unsigned long quadraticValueForX = PerformProjectionFor_X(
        z_xCoordinatePoint);

        // encoding process
        unsigned int iV, oV, rV;

        {   unsigned int result1, result2, result3, result4;
            iV=(bit_from_stream);

               // from the iteration of the 'y' quadratic
            result1=quadraticValueForY;

               // from the iteration of the 'x' quadratic
            result2=quadraticValueForX;

               // we will copy the bits if it is set in one operand but not
                  both
            result3=iV^result1;
            result4=iV^result2;

               // the output value
            oV=result3;
            rV=result4; // branch in path to follow through IIM
        }
        bit_from_coding=(oV);

        if ((rV) != 0)
        {
               // use the route on the second root point
            z_xCoordinatePoint=-z_xCoordinatePoint;
            z_yCoordinatePoint=-z_yCoordinatePoint;
        }
    }
    return bit_from_coding;
};
```

```
unsigned int CryptoFractalCipher::bitCodeDecryptedMessageD(unsigned int
bit_from_stream)
{
     // generate the cryptographic stream from the clear stream
    int bit_from_coding = 0;
    {
        identifyFirstRoot();

        unsigned long quadraticValueForY = PerformProjectionFor_X(
        z_yCoordinatePoint);
        unsigned long quadraticValueForX = PerformProjectionFor_X(
        z_xCoordinatePoint);

        // encoding process
        unsigned int iV, oV, rV;
        {
               unsigned int result1, result2, result3, result4;
            iV=(bit_from_stream) & 1;

               // from iterated quadratic y and x
            result1=quadraticValueForY;
            result2=quadraticValueForX;

               // we will copy the bits if it is set in one operand but not
                  both
            result3=iV^result1;
            result4=result3^result2;

               // output value
            oV=result3;

               // the route value
            rV=result4;
        }
        bit_from_coding=(oV);
```

```
        if ((rV) != 0)
        {   //take branch to second root
            z_xCoordinatePoint=-z_xCoordinatePoint;
            z_yCoordinatePoint=-z_yCoordinatePoint;
        }
    }
    return bit_from_coding;
};

unsigned int CryptoFractalCipher::getEncryptedMessageA(unsigned int
clearstream)
{
     // for creating the cryptographic stream from the clear stream
    int cryptographic_stream=0;

    for (int iterationIndex=0; iterationIndex<32; (iterationIndex++))
    {

        // encoding process for obtaining cryptographic stream from clear
           stream
        unsigned int iV,oV;
        iV=(clearstream>>iterationIndex) & 1;
        oV=bitCodeEncryptedMessageA(iV);
        cryptographic_stream+=((oV)<<iterationIndex);
    }

    return cryptographic_stream;
};

unsigned int CryptoFractalCipher::getDecryptedMessageB(unsigned int
cryptstream)
{
     // for creating the clear stream from the cryptographic stream
    int clearstream=0;

    for (int iterationIndex=0; iterationIndex<32; (iterationIndex++))
    {
```

```
          // decoding process for obtaining the clear stream from the
             cryptographic stream
        unsigned int iV, oV;

        iV=(cryptstream>>iterationIndex) & 1;
        oV=bitCodeDecryptedMessageB(iV);
        clearstream+=((oV)<<iterationIndex);
    }
    return clearstream;
};

unsigned int CryptoFractalCipher::getEncryptedMessageC(unsigned int stream)
{
     // construct the cryptographic stream from clear stream
     // cv - the code value
    int cV=0;

    for (int iterationIndex=0; iterationIndex<32; (iterationIndex++))
    {
          // encoding process for generating the cryptographic stream from
             clear stream
        unsigned int iV,oV;
        iV=(stream>>iterationIndex) & 1;
        oV=bitCodeEncryptedMessageC(iV);
        cV+=((oV)<<iterationIndex);
    }
    return cV;
};

unsigned int CryptoFractalCipher::getDecryptedMessageD(unsigned int stream)
{
     // construct the cryptographic stream from clear stream
     // cv - the code value
    int cV=0;

    for (int iterationIndex=0; iterationIndex<32; (iterationIndex++))
    {
```

```
        // encoding process for generating the cryptographic stream from
           clear stream
        unsigned int iV, oV;
        iV=(stream>>iterationIndex) & 1;
        oV=bitCodeDecryptedMessageD(iV);
        cV+=((oV)<<iterationIndex);
    }
    return cV;
};

CryptoFractalCipher::CryptoFractalCipher(double cPoint_xValue,double
cPoint_yValue)
{
    c_xCoordinatePoint=cPoint_xValue;
    c_yCoordinatePoint=cPoint_yValue;

    z_xCoordinatePoint=z_yCoordinatePoint=0;

     // use repeating digits as for encoding process using PI value with
        the goal to find a fixed point
    for(int index=0; index<32; index++)
          getEncryptedMessageA(3141592653);
}

// destructor implementation - only if it is necessary
CryptoFractalCipher::~CryptoFractalCipher()
{
}

int main(void)
{
     // CryptoKey_rValue and CryptoKey_iValue are represented as
     // a point that is situated near the boundary of the Mandelbrot set
     // the real value of a complex number (cryptographic key)
     double CryptoKey_rValue=-0.687;

     // the imaginary unit
     double CryptoKey_iValue=-0.312;
```

```
unsigned int Plaintext[50];
unsigned int EncryptionA[50];
unsigned int EncryptionB[50];
unsigned int DecryptionOfAWithB[50];
unsigned int DecryptionOfBWithA[50];

// generate randomly message
for (int i=0;i<50;i++)
     Plaintext[i]=rand()%1000;

// perform message encoding using getEncryptedMessageA for A
{
     CryptoFractalCipher CFC(CryptoKey_rValue, CryptoKey_iValue);
     for (int i=0;i<50;i++)
        EncryptionA[i]=CFC.getEncryptedMessageA(Plaintext[i]);
}

// perform message encoding using getDecryptedMessageB for B
{
     CryptoFractalCipher CFC(CryptoKey_rValue, CryptoKey_iValue);
     for (int i=0;i<50;i++)
        EncryptionB[i]=CFC.getDecryptedMessageB(Plaintext[i]);
}

// perform message decoding A with B using getDecryptedMessageB for B
{
     CryptoFractalCipher CFC(CryptoKey_rValue, CryptoKey_iValue);
     for (int i=0;i<50;i++)
        DecryptionOfAWithB[i]=CFC.getDecryptedMessageB(EncryptionA[i]);
}
// perform message decoding B with A using getDecryptedMessageB for A
{
     CryptoFractalCipher CFC(CryptoKey_rValue, CryptoKey_iValue);
     for (int i=0;i<50;i++)
        DecryptionOfBWithA[i]=CFC.getEncryptedMessageA(EncryptionB[i]);
}
```

```
    // display the output value and the results
    for (int i=0;i<50;i++)
    {
          cout
          <<i
          <<")   (Plaintext Value="<<Plaintext[i]
          <<")   (Encryption -> First Method (A) = "<<EncryptionA[i]
          <<")   (Encryption -> Second Method (B) = "<<EncryptionB[i]
          <<")   (Decryption -> A with B = "<<DecryptionOfAWithB[i]
          <<")   (Decryption -> B with A = "<<DecryptionOfBWithA[i]
          <<")"<<endl;
    };
}
```

Conclusion

In this chapter, we discussed a different approach in cryptography, which is chaos-based cryptography. The new cryptographic algorithms use the chaos function to generate new cryptographic primitives in a different way from the ones we know so well.

At the end of this chapter, you will know the following:

- How chaos-based cryptography primitives are built and what makes them different from the normal cryptographic primitives
- How the chaos system is designed for text encryption and image encryption
- How to implement a cryptographic system based on number generators using the chaos approach and how to perform encryption and decryption operations with the chaos system and fractals

References

[1] Robert Matthews, “On the derivation of a ‘chaotic’ encryption algorithm” in *Cryptologia 13, no. 1* (pp. 29-42). 1989.

[2] L. Kocarev, "Chaos-based cryptography: A brief overview" in *IEEE Circuits and Systems Magazine, vol. 1, no. 3* (pp. 6-21). doi: 10.1109/7384.963463. 2001.

[3] Ali Soleymani, Zulkarnain Md Ali, and Md Jan Nordin, "A Survey on Principal Aspects of Secure Image Transmission" in *World Academy of Science, Engineering and Technology 66* (pp. 247–254). 2012.

[4] D. Chattopadhyay1, M. K. Mandal1, and D. Nandi, "Symmetric key chaotic image encryption using circle map" in *Indian Journal of Science and Technology, Vol. 4 No. 5* (pp 593–599). ISSN: 0974-6846. May 2011.

[5] A. Anto Steffi and Dipesh Sharma, "Modified Algorithm of Encryption and Decryption of Images using Chaotic Mapping" in *International Journal of Science and Research (IJSR), India Online Volume 2 Issue 2*. ISSN: 2319-7064, February 2013.

[6] K. Sakthidasan Sankaran and B.V. Santhosh Krishna, "A New Chaotic Algorithm for Image Encryption and Decryption of Digital Color Images" in *International Journal of information and Education Technology, Vol. 1, No. 2*. June 2011.

[7] Somaya Al-Maadeed, Afnan Al-Ali, and Turki Abdalla, "A New Chaos-Based Image-Encryption and Compression Algorithm" in *Journal of Electrical and Computer Engineering, Volume 2012*. Hindawi Publishing Corporation. Article ID 179693. 2012.

[8] Hazem Mohammad Al-Najjar and Asem Mohammad AL-Najjar, "Image Encryption Algorithm Based on Logistic Map and Pixel Mapping Table."

[9] Sodeif Ahadpour and Yaser Sadra, "A Chaos-based Image Encryption Scheme using Chaotic Coupled Map Lattices."

[10] Kamlesh Gupta1 and Sanjay Silakari, "New Approach for Fast Color Image Encryption Using Chaotic Map" in *Journal of Information Security 2* (pp. 139-150). 2011.

[11] Chong Fu, Jun-jie Chen, Hao Zou, Wei-hong Meng, Yong-feng Zhan, and Ya-wen, "A chaos-based digital image encryption scheme with an improved diffusion strategy" in *Optical Society of America, Vol. 20, No. 3* (pp. 2363–2378). January 30, 2012.

[12] D. Chattopadhyay1, M. K. Mandal1, and D. Nandi, "Symmetric key chaotic image encryption using circle map" in *Indian Journal of Science and Technology, Vol. 4 No. 5* (pp. 593–599). ISSN: 0974-6846. May, 2011.

[13] Shima Ramesh Maniyath1 and Supriya M, "An Uncompressed Image Encryption Algorithm Based on DNA Sequences" in *Computer Science & Information Technology (CS & IT)*, CCSEA 2011, CS & IT 02 (pp. 258–270). 2011.

[14] Miguel Murillo-Escobar, "A novel symmetric text encryption algorithm based on logistic map." 2014.

[15] K. Sakthidasan and B. V. Santhosh Krishna. "A New Chaotic Algorithm for Image Encryption and Decryption of Digital Color Images" in *International Journal of Information and Education Technologies, vol. 1, no. 2*, June 2011. Available online: www.ijiet.org/papers/23-E20098.pdf.

[16] Ljupco Kocarev and Shinguo Lian. *Chaos-based Cryptography – Theory, Algorithms, and Applications*. Springer, 2011.

[17] Heinz-Otto Peitgen, Hartmut Jurgens and Dietmar Saupe. *Fractals for Classroom, Part Two – Complex Systems and Mandelbrot Set*. Springer-Verlag, 1992.

CHAPTER 15

Big Data Cryptography

Big data can be seen as the processes through which data sets of a big size (in a range from a few terabytes to many zettabytes) are extracted, manipulated, and analyzed. These techniques differ from traditional techniques because big data contains different types of data, structured or unstructured (video or audio files, images, texts, etc.).

Big data cryptography is related to data's confidentiality, integrity, and authenticity, representing an important topic that needs to be treated with attention because each business has its computational model and software and hardware architecture. The cryptographic methods related to big data differ from the traditional ones because encryption systems and their related concepts are defined differently regarding the policies for access control, cloud infrastructure, and storage management and techniques.

This chapter starts by describing a general computational model applicable in a cloud environment that enables and eases the implementation of applications that involve big data analytics. In the following, we present a classification of the nodes from the cloud architecture and their purpose in the big data analytics process. The types of nodes are based on the classification from [1] - [3] and the notations are extended a little to define the following types of nodes:

- I_N represents an *input node* that handles the raw data used in the application. These types of nodes collect the data from the front-end users or the data that is read or captured from different sensors (such as fingerprint readers, holographic signatures, temperature sensors, etc.).
- C_N represents the *computational node,* which has a significant role in the computational processes from the application. The basis of these nodes is the ingestion nodes, which are included in a computational node. In this classification, the ingestion nodes are called *consuming nodes* and their purpose is to scan and refine the input data, meaning data preparation for the analysis process and its passing to the *enrichment nodes*, where the data is actually processed.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_15

- S_N represents the *storage node,* which has a significant role in applying the cryptographic techniques over the data. Its purpose is to store the data involved in the computational processes that are applied between different types of end users and third parties. The input data and the output data for data analysis are stored by these nodes.
- R_N represents the *result node,* which receives the output of some processes that are being executed. It can make automatic decisions based on the output of the analysis process or it can send the output to a specific client.

Figure 15-1 shows an example of cloud architecture for big data analytics that includes the elements described above. The model can represent a pattern that describes a wide range of big data applications. This being said, we will note the following set of one or more nodes of type H, as follows H^+, where $H \in \{I_N, C_N, S_N, R_N\}$.

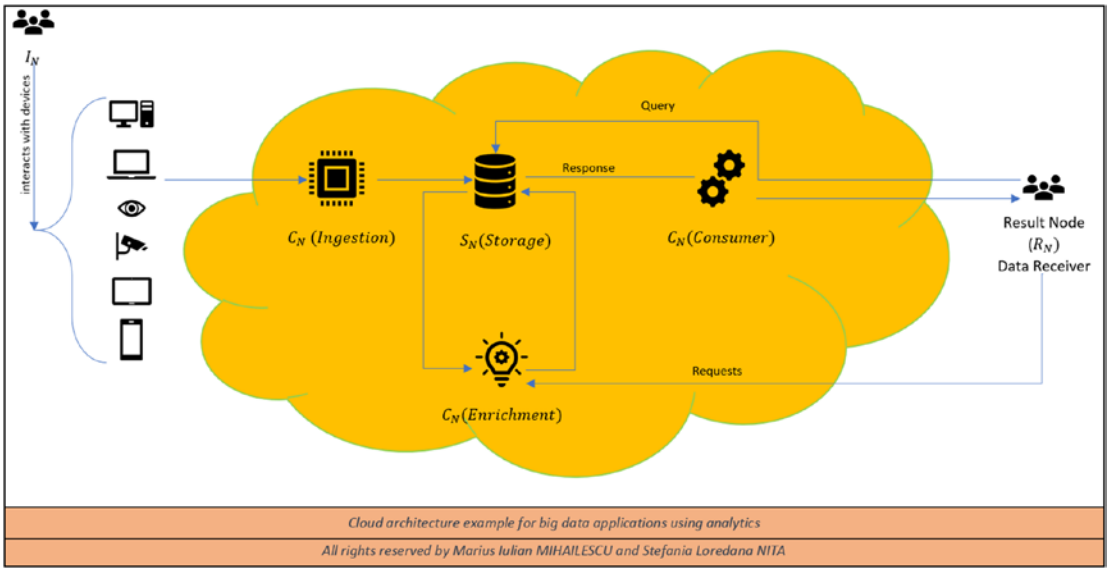


Figure 15-1. *Example of cloud architecture with big data analytical applications*

Figure 15-1 shows a general cloud model that can be applied to an application that requires data sets. In the example, the node I_N initiates the process of collecting reference datasets. The input nodes send sequences of data to the C_N(*Ingestion*) node. In the ingestion node, the sequences of data are used by the computation process

for which they are parsed. When the computational process ends, the output data is organized in files or databases. In the next step, the files and databases are sent to the storage nodes S_N (*Storage*). From time to time, the enrichment nodes C_N(*Enrichment*) perform computation over the data from the storage nodes. Mostly, these processes are made offline and update the associated metadata according to the user's needs. In our example, the R_N (*Data Receiver*) represents a user who will correlate the data set with the reference one.

Cloud computing presents many security challenges for the data that moves through and between its components. To follow the path of protection techniques from the cloud cryptography, we need to take into consideration three main security goals, known as the CIA triad:

- **Confidentiality**: The data refers strictly to the input and output of the computations and needs to be kept secret and protected against untrusted parties, malicious parties, or other potential adversaries.
- **Integrity**: Any changes that are not authorized over the data must be immediately detected. Note that integrity issues are not always caused by nefarious actors; they can also be caused by bugs in software or issues in data transfers. Regardless, the integrity of the data must be enforced.
- **Availability**: The data owners and the authorized data users have access to the data and computational resources.

Let's focus on availability because it is one of the most important characteristics for the cloud, but it does not include any cryptographic means. For this reason, confidentiality and integrity need to be involved as much as possible in the cloud and big data architecture. The way in which data is stored is relevant, too, for security and cryptographic purposes. The way in which confidentiality and integrity are achieved is dictated by the way in which the cloud is deployed. When developing an application, it is important to establish from the beginning which participant controls which component of the cloud and the degree of trust awarded to each component and participant. Based on this, we will consider the following types of clouds:

- **Trusted cloud**: It is deployed by government organizations or institutions, and it is isolated completely from anything from outside (networks or adversaries). Public cloud vendors such as Microsoft have regions for US government users. The Microsoft Jedi contract

with the DOD covers such use as well as Azure cloud resources authorized for secret and top secret use. The files of the users or clients are stored safely, without any worry of corruption or theft. However, there are situations in which some of the nodes are exposed because they may communicate with external networks. Therefore, in these situations, malware or insiders can affect these types of nodes.

- **Semi-trusted cloud**: In this type of cloud, it is not mentioned specifically if the cloud can be trusted entirely or cannot be trusted at all. However, a good practice is to mention the components that are under control and to provide solutions to monitor the adversarial activities at a given time.
- **Untrusted cloud**: The nodes within the cloud or the cloud itself are not trusted at all by the data users. This scenario means that no security guarantees are given, including a level of confidentiality or integrity of the data or computations. In such situations, the cloud user should have its own solutions and protection mechanisms to ensure (a level of) confidentiality and integrity. Mainly, the untrusted cloud is associated with the public cloud model.

With a short description of the cloud and big data elements, we can go further to discuss the *cryptographic techniques* that can be applied in these environments. To ensure the security of big data and cloud computing, cryptographic techniques are very complex and it is difficult to apply them in real-life scenarios without dedicated third-party software libraries or experienced professionals.

This chapter focuses on three cryptographic techniques that can be used particularly for achieving security of big data applications deployed in the cloud environment, such as

- Homomorphic encryption (HE): See Chapter 12
- Verifiable computation (VC): Represents the first objective of this chapter
- Secure multi-party computation (MPC)

Other cryptographic techniques that can be applied successfully to achieve security in cloud computing and big data are

- Functional encryption (FE)

- Identity-based encryption (IE)
- Attribute-based encryption (AE)

In the next section, we present a technique that is promising and can be applied in real environments. Many of the encryption schemes that fall in the FE, IE, or AE types are very difficult to use in practice because many works are based on theoretical assumptions and most of them don't take into consideration the requirements and demands of business or industry applications. Between theory and practice, it is a long path that theoreticians and practitioners need to walk together. They need to collaborate closely in order to find solutions for the security concerns in real environments and to solve the problems and gaps that exist.

Verifiable Computation

Verifiable computation or *verifiable computing* refers to the machines' capability of unloading the computation quantity of some function(s) to others, for example, clients with untrusted status, while the results are verified continuously. See Figure 15-2.

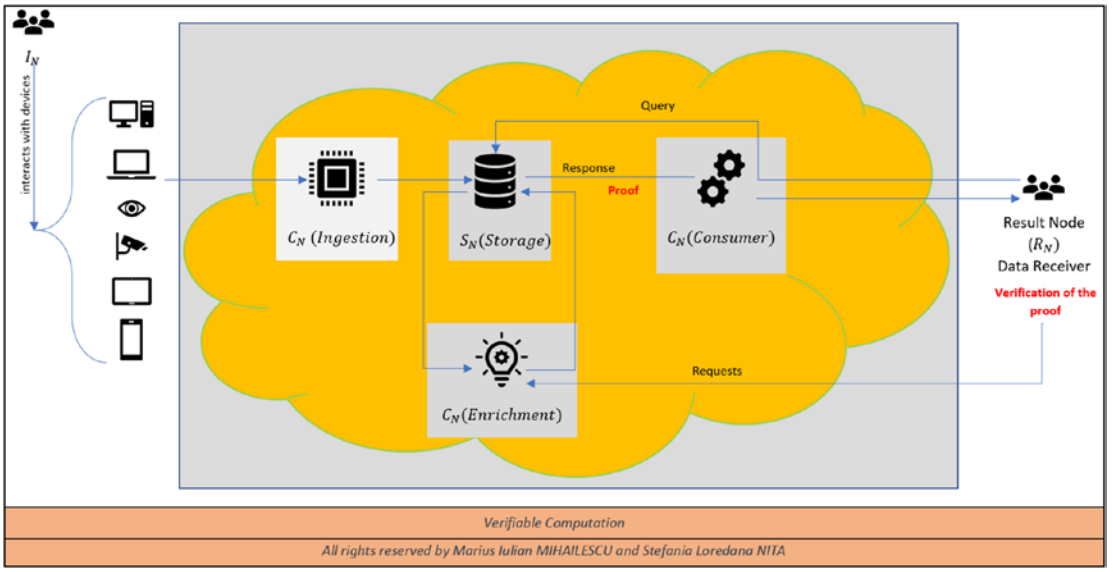


Figure 15-2. *Verifiable computation example. The nodes of the cloud don't have any trustiness level for integrity protection*

An important application of verifiable computation for real environments are Merkle trees, whose purpose is to check the integrity of the data. In big data, the Merkle tree represents a data structure that is used for validation of the integrity of different properties for items, data, rows, sets of data, and so on. A very useful characteristic of a Merkle tree is that it can be used on large amounts of data (therefore, in the context of big data) and in this direction many improvements have been made by combining algorithms of verifiable computation with Merkle trees.

In Listing 15-1 through Listing 15-6, we present a scenario in which a Merkle tree is self-balancing. The example is just a simulation (see Listing 15-1, Listing 15-6, and Figure 15-3). Deploying the application in a real big data environment will require proper adjustments.

The code is organized in the following files:

- `tree_node.cpp` contains the implementation of the methods used with a tree node.
- `tree_node.h` contains the definitions of a tree node.
- `tree.cpp` contains the implementations of the methods used with a tree.
- `tree.h` contains the definitions of a tree.
- `tree_handling.h` contains the function of printing and computing the `sha256` value of the information within a node.
- `picosha2.h` is downloaded as is from the source [4] and represents a header file for computing the sha256 hash value of an input. Its content can be found in the source [4] or in this chapter's code folder on the GitHub repository for the book.
- `main.cpp` is the main file of the project.

Listing 15-1. The Content of the tree_node.h File

```
#ifndef TREE_NODE
#define TREE_NODE

#include <string>
using namespace std;
```

```
// define the node of the merkle tree
struct tree_node
{
    string hash_value; // the hash value
    tree_node *l_neighbour; // the left neighbour
    tree_node *r_neighbour; // the right neighbour

// instantiates the hash value within the node
// see the corresponding .cpp file
    tree_node(string value);
};

#endif
```

Listing 15-2. The Content of the tree_node.cpp File

```
#include "tree_node.h"
using namespace std;

// assigns the input hash value to the hash_value attribute of the tree node
tree_node::tree_node(string value)
{
    this->hash_value = value;
}
```

Listing 15-3. The Content of the tree.h File

```
#ifndef MERKLE_TREE
#define MERKLE_TREE

#include "tree_node.h"
#include "picosha2.h"
#include "tree_handling.h"
#include <vector>
#include <string>

using namespace std;
```

```
struct merkle_tree {
    tree_node* tree_root;
    merkle_tree(vector<tree_node*> vector_nodes);
    ~merkle_tree();
    void print_merkle_tree(tree_node *node, int index);
    void delete_merkle_tree(tree_node *node);
};

#endif
```

Listing 15-4. The Content of the tree.cpp File

```
#include <iostream>
#include <iomanip>
#include "tree.h"

using namespace std;

merkle_tree::merkle_tree(vector<tree_node*> vector_nodes)
{
    vector<tree_node*> aux_nodes;
    while (vector_nodes.size() != 1)
    {
        print_hash_values(vector_nodes);
        for (int i = 0, n = 0; i < vector_nodes.size(); i = i + 2, n++) {
            if (i != vector_nodes.size() - 1) // check if there is a
            neighbour block
            {
                // merges the neighbour nodes and computes the hash value
                   of the new node
                aux_nodes.push_back(new tree_node(compute_sha256(vector_
                nodes[i]->hash_value + vector_nodes[i + 1]->hash_value)));
                // link the new node with the left neighbour and the right
                   neighbour
                aux_nodes[n]->l_neighbour = vector_nodes[i];
                aux_nodes[n]->r_neighbour = vector_nodes[i + 1];
            } else
```

```
            {
                aux_nodes.push_back(vector_nodes[i]);
            }
        }
        cout << "\n";
        vector_nodes = aux_nodes;
        aux_nodes.clear();
    }

    // picks the first node as the root of the tree
    this->tree_root = vector_nodes[0];
}

merkle_tree::~merkle_tree()
{
    delete_merkle_tree(tree_root);
    cout << "The tree was deleted." << endl;
}

void merkle_tree::print_merkle_tree(tree_node *node, int index)
{
    if (node) {
        if (node->l_neighbour) {
            print_merkle_tree(node->l_neighbour, index + 4);
        }
        if (node->r_neighbour) {
             print_merkle_tree(node->r_neighbour, index + 4);
        }
        if (index) {
            cout << setw(index) << ' ';
        }
        cout << node->hash_value[0] << "\n ";
    }
}
```

```
void merkle_tree::delete_merkle_tree(tree_node *node)
{
    if (node) {
        delete_merkle_tree(node->l_neighbour);
        delete_merkle_tree(node->r_neighbour);
        node = NULL;
        delete node;
    }
}
```

Listing 15-5. The Content of the tree_handling.h File

```
#ifndef TREE_MISC
#define TREE_MISC

#include <iostream>
#include <string>
#include "tree.h"
#include "picosha2.h"

using namespace std;

// computes the hash value of the input using SHA256
inline string compute_sha256(string input_string)
{
    string hash_string = picosha2::hash256_hex_string(input_string);
    return hash_string;
}

// display the hash values from a vector of tree nodes
inline void print_hash_values(vector<tree_node*> vector_nodes)
{
    for (int i = 0; i < vector_nodes.size(); i++)
    {
        cout << vector_nodes[i]->hash_value << endl;
    }
}

#endif
```

Listing 15-6. The Content of the main.cpp File

```
#include <iostream>
#include "tree.h"

using namespace std;

int main() {
    vector<tree_node*> nodes_set;

    //create sample data
    nodes_set.push_back(new tree_node(compute_sha256("Merkle ")));
    nodes_set.push_back(new tree_node(compute_sha256("tree ")));
    nodes_set.push_back(new tree_node(compute_sha256("node ")));
    nodes_set.push_back(new tree_node(compute_sha256("example.")));
    nodes_set.push_back(new tree_node(compute_sha256("This is an example of
    merkle tree.")));

    // initialize leaves
    for (unsigned int i = 0; i < nodes_set.size(); i++) {
        nodes_set[i]->l_neighbour = NULL;
        nodes_set[i]->r_neighbour = NULL;
    }

    merkle_tree *hash_tree = new merkle_tree(nodes_set);
    std::cout << hash_tree->tree_root->hash_value << std::endl;
    hash_tree->print_merkle_tree(hash_tree->tree_root, 0);

    for (int k = 0; k < nodes_set.size(); k++) {
        delete nodes_set[k];
    }

    delete hash_tree;

    return 0;
}
```

To compile the code, the following command is used in the terminal:

```
g++ -o result.exe main.cpp tree_node.cpp tree_node.h tree.cpp tree.h
```

To run the code, type in the terminal:

```
result
```

The result can be seen in Figure 15-3.

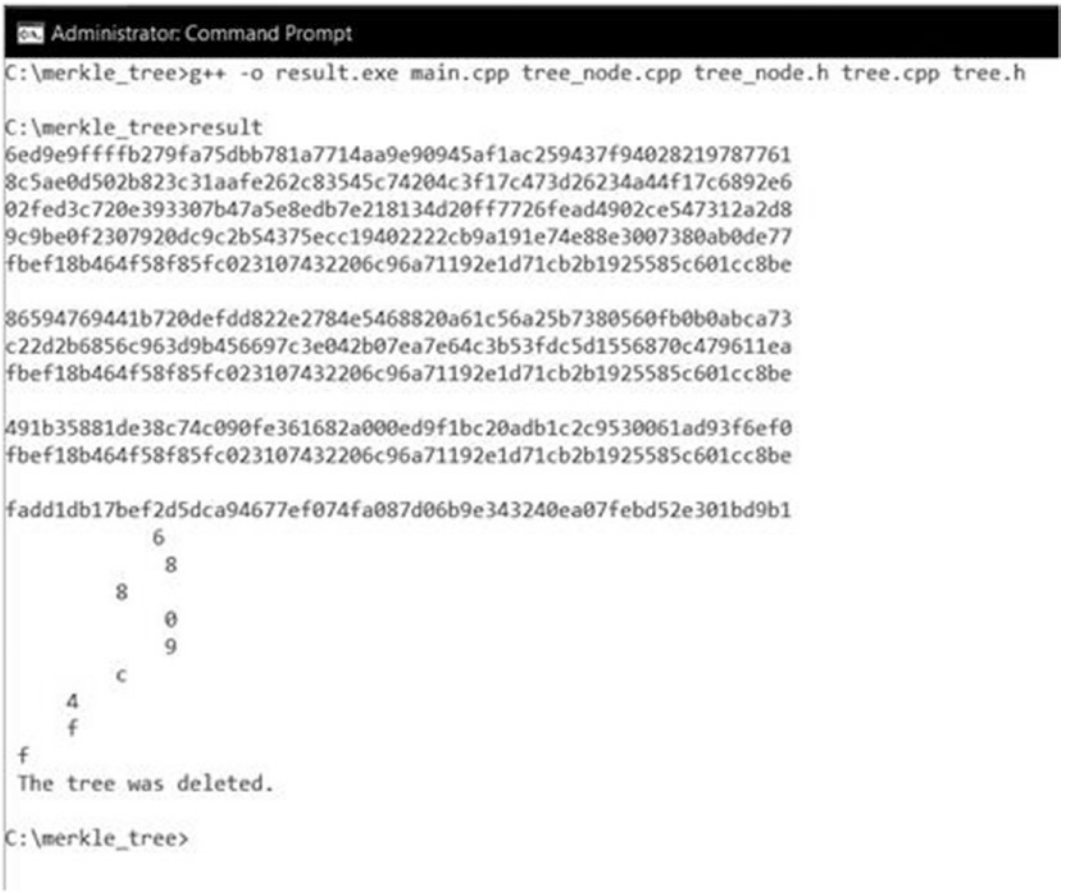


Figure 15-3. *The result of the implementation of a self-balancing Merkle tree*

Conclusion

In this chapter, we discussed the importance of an application deployed in the big data environment and the way in which security can be achieved through different cryptographic mechanisms, such as verifiable computation. For more about cloud computing, big data, and security, you may consult any of the works from this chapter's references.

At the end of this chapter, you will have the following knowledge:

- Understanding the main concepts of security in a cloud and big data environment
- How to put into practice complex cryptographic primitives and protocols, such as verifiable computation

References

[1] P. Laud and A. Pankova, “Verifiable Computation in Multiparty Protocols with Honest Majority” in (eds. S.S.M. Cho, J.K. Liu, L.C.K. Hui, and S.M. Yiu) *Provable Security. ProvSec 2014. Lecture Notes in Computer Science, vol 8782*. Springer, Cham. 2014.

[2] D. Bogdanov, S. Laur, and R. Talviste, “A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation” in (eds. K. Bernsmed and S. Fischer-Hübner) *Secure IT Systems. NordSec 2014. Lecture Notes in Computer Science, vol 8788*. Springer, Cham. 2014.

[3] D. Bogdanov, L. Kamm, S. Laur, and P. Pruulmann-Vengerfeldt, “Securemulti-party data analysis: End user validation and practical experiments,” 2014.

[4] PicoSHA2 - a C++ SHA256 hash generator. Available online: https://github.com/okdshin/PicoSHA2

[5] B. ÖzÇakmak, A. Özbİlen, U. YavanoĞlu, and K. Cİn, "Neural and Quantum Cryptography in Big Data: A Review" in *2019 IEEE International Conference on Big Data (Big Data)* (pp. 2413-2417). Los Angeles, CA, USA. doi: 10.1109/BigData47090.2019.9006238. 2019.

[6] S. Yakoubov, V. Gadepally, N. Schear, E. Shen, and A. Yerukhimovich, “A survey of cryptographic approaches to securing big-data analytics in the cloud,” in *2014 IEEE High Performance Extreme Computing Conference (HPEC) (pp.106).* Waltham, MA. doi: 10.1109/HPEC.2014.7040943. 2014.

[7] S.L. Nita and M.I. Mihailescu, "A Searchable Encryption Scheme Based on Elliptic Curves" in (eds. L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa) *Web, Artificial Intelligence and Network Applications. WAINA 2020. Advances in Intelligent Systems and Computing, vol 1150*. Springer, Cham. 2020.

[8] S.L. Nita and M.I. Mihailescu, "A Hybrid Searchable Encryption Scheme for Cloud Computing" in (eds. J.L. Lanet and C. Toma) *Innovative Security Solutions for Information Technology and Communications. SECITC 2018. Lecture Notes in Computer Science, vol 11359*. Springer, Cham. 2019.

[9] V. C. Pau and M. I. Mihailescu, "Internet of Things and its role in biometrics technologies and eLearning applications" in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, (pp. 1-4). Oradea. doi: 10.1109/EMES.2015.7158430. 2015.

[10] S. L. Nita and M. I. Mihailescu, "On Artificial Neural Network used in Cloud Computing Security - A Survey" in *2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, (pp. 1-6). Iasi, Romania. doi: 10.1109/ECAI.2018.8679086. 2018.

[11] Marius Iulian Mihailescu, Stefania Loredana Nita, and Ciprian Racuciu, "Authentication protocol based on searchable encryption and multi-party computation with applicability for earth sciences" in *Scientific Bulletin of Naval Academy, Vol. XXIII* (pg.221-230), doi: 10.21279/1454-864X-20-I1-030. 2020.

[12] Marius Iulian Mihailescu, Stefania Loredana Nita, and Ciprian Racuciu, "Multi-level access using searchable symmetric encryption with applicability for earth sciences" in *Scientific Bulletin of Naval Academy, Vol. XXIII* (pp.221-230). doi: 10.21279/1454-864X-20-I1-030. 2020.

[13] Stefania Loredana Nita, Marius Iulian Mihailescu, and Ciprian Racuciu, "Secure Document Search in Cloud Computing using MapReduce" in *Scientific Bulletin of Naval Academy, Vol. XXIII* (pp. 221-230). doi: 10.21279/1454-864X-20-I1-030. 2020

[14] RSA Extension for Big Data Analytics. Available online: www.rsa.com/en-us/company/news/rsa-extends-big-data-analytics-to-help-organizations-identify.

[15] Claudio Orlandi, "Is Multiparty Computation Any Good In Practice?" Available online: www.cs.au.dk/~orlandi/icassp-draft.pdf.

CHAPTER 16

Cloud Computing Cryptography

Cryptography in cloud computing has gained a lot of attention in the few past years. Nowadays it's one of the most important topics in cryptography and cybersecurity. It represents a key point in the design and implementation of a secure cloud application. Cryptography for cloud computing involves complex encryption methods and techniques for securing data that is stored and used in the cloud environment.

There are three main types of cloud technologies that organizations have adopted rapidly: IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service), and SaaS (Software-as-a-Service). The cloud offers many benefits for its users, such as efficiency, flexibility, and scalability, which lead to reducing the overall cost for the clients. Due to its complexity and types (public, private, or hybrid cloud), cloud computing inherits the security concerns of its components. Source [1] provides a great categorization of cloud computing security issues. The security concerns may occur on the following levels: the communication level (which deals with the shared infrastructures, virtual networks, and their configurations), the architectural level (which deals with virtualization, data storage, applications and APIs, and access control) and even the contractual and legal level (which deals, for example, with service level agreements).

To secure the cloud, the following cryptographic techniques and mechanisms are receiving important attention from research communities and industries:

- Searchable encryption (see Chapter 11)
- Homomorphic encryption (see Chapter 12)
- Structured encryption (STE), which is used to encrypt the data structures. An STE scheme uses a token to query the data structure. A special example of STE is searchable encryption (SE). Recall that

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_16

searchable encryption allows for searching for a keyword through data in an encrypted format. Another example of STE is using graph structures to encrypt databases. It is a good example in the cloud context, where applications deal with large databases for analytics and statistics.

- Functional encryption (FE) can be considered a generalization of the public-key encryption, where the owner of the private key allows an authorized user to learn a function of what ciphertext is being encrypted. There are more types of functional encryption: predicate encryption (PE), identity-based encryption (IBE), attribute-based encryption (ABE), hidden vector encryption (HVE), and inner product predicate.
- Private information retrieval (PIR) is actually a protocol used by a client to retrieve an element within a database without letting the rest of the database users know what element the client retrieved.

A Practical Example

For this example, let's imagine the following cloud scenario: an organization manages its administrative relationship with its clients using a cloud messaging platform. For example, the organization sends notifications to their clients about their products or available updates, and the clients can send messages to the organization through the platform. Therefore, the cloud platform is included in the category Software-as-a-Service. To ensure that the messages are read by the authorized receiver, the messages should be encrypted from both sides, the organization and the clients, and both of them should use trusted parties for the key generation that will be used in the encryption and decryption.

To simulate this example, we will use as a trusted party OpenSSL [1], which will generate the public and the private keys for the RSA algorithm, keys that will be used in our encryption technique. Source [1] provides documentation for different distributions, links to source code from a GitHub repository, examples, and much more. Note that we created this example on a Windows platform. You will not download the source code to compile it yourself and then use it. Instead, you will download directly the compiled version of the OpenSSL library that can be found at source [3] (or it can be downloaded from the GitHub repository of this book). Once the archive is downloaded, you extract

it to the OpenSSL folder on the `C:\` partition. Further, you open a terminal and change the current directory to the `bin` folder from the OpenSSL parent folder and then type `openssl` and press Enter (Figure 16-1).

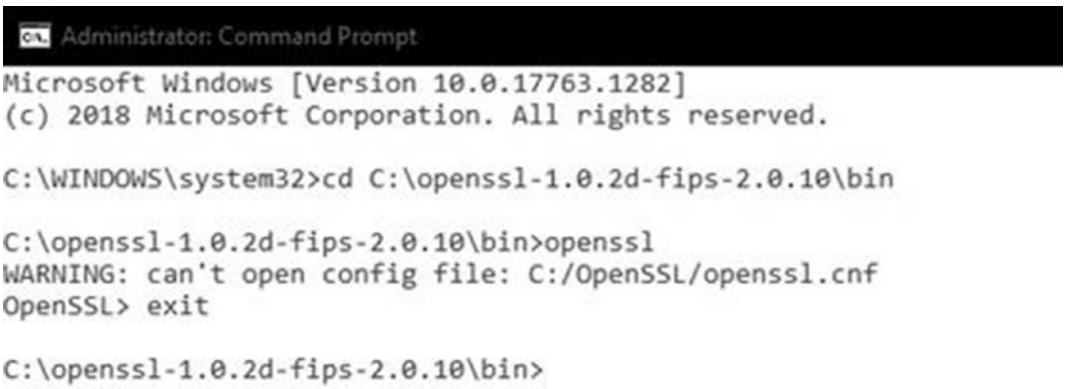


Figure 16-1. *Checking the openssl command*

The message warning shows because we used the OpenSSL package and it is not compiled on the computer. For the purpose of this section, you do not need to compile and install OpenSSL yourself, but the complete guide for installing it can be found in the source [2]

The next step is to generate the private key for the RSA algorithm. To do this, type the following command in the terminal and check Figure 16-2:

```
C:\openssl-1.0.2d-fips-2.0.10\bin>openssl genrsa -out privateKey.pem 2048
```

The above command says that the `openssl` library is used to generate the RSA private key (`genrsa`) in the output file `privateKey.pem`, having a length of `2048` bits.

Then, to generate the public key type, the following command in the terminal and check Figure 16-2:

```
C:\openssl-1.0.2d-fips-2.0.10\bin>openssl rsa -in privateKey.pem -pubout >
publicKey.pem
```

The above command says that the `openssl` library is used to compute the public key of the cryptosystem saved in the output file `publicKey.pem`, based on the input file (private key) `privateKey.pem`.

```
Administrator: Command Prompt
C:\openssl-1.0.2d-fips-2.0.10\bin>openssl genrsa -out privateKey.pem 2048
WARNING: can't open config file: C:/OpenSSL/openssl.cnf
Loading 'screen' into random state - done
Generating RSA private key, 2048 bit long modulus
.....................................................................................+++
.....................................+++
e is 65537 (0x10001)

C:\openssl-1.0.2d-fips-2.0.10\bin>openssl rsa -in privateKey.pem -pubout > publicKey.pem
WARNING: can't open config file: C:/OpenSSL/openssl.cnf
writing RSA key

C:\openssl-1.0.2d-fips-2.0.10\bin>
```

Figure 16-2. *Generating the private key and the public key for the RSA cryptosystem*

The files `publicKey.pem` and `privateKey.pem` are generated in the same folder as the `openssl` library is, namely the `bin` folder. If you check the contents of these files, they should look like Figure 16-3a and 16-3b.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEA4QYm+aUBDY2hZ8RLLdmvCtgeGjGhg0A5+9nITcAjFEWQ9ELT
OA8iO+k8EO+ATJWMN5+/2xj1QZgvVpVx83MOl03XopBWYZvCljftWaBsCX97NI7l
tadjNnO5KnIboSV/YqMIBRzVCOFmtF756KlWcBMj0pb/M5IcmcfTWHZAt7lNYueP
tLDuVrWc/M3YALJycsSaicFH3ecp9SP7bhg3VgMOueGllc3KFgqfyZitD6TJAYrG
Fx4CS/IjCo+/4odjNeGXL47KIDOGm3X7XnMHtu86IqWLIM0m7xsmRR5FT/qBAjCf
2/BrElofJPff2YkeZbFA9BtpjUWVrsK4vZKtzwIDAQABAoIBAGKhxberIZkFOXg4
TBOe1ODV9evZUUgsgL3pqaiazUV3cKAAIoDx8Tl9c/ygXgYzjrBa70dj2yyTPjyf
aMEi74RWdGwnBBXopjArd3fPn0tSe3iAIiPLaOtXMhRuy7mIrBtaBovnJWGtGsl4
y+WlQ9QAL7WAxf+ezbaCJhsiSh+84MSgyProkKAxMMOFttlpJ/JS5LAUpXpNNDRi
+mXMbR2P5Jf3pGWPAty/psRw2PcX1PTQkQEXEU77S+fv4hKle4b0oO+bdV4C55GF
lGp9QXbIbPEflRhJx7GpggZnxrAk/T0oUtDtp8Rq6zcgbumVF8WlI+DjEElOl9et
q7TGpqECgYEA9CzazwnJquBaJRItvILn/PCi/Wtk5VVeEZc6UgDtYDmylyoGp8nL
xUC4gPS69MPl5QaQQajB0HBhNGcc4WwejTYyWO/teNHMBOEPYyK3xK4g/QIEb2ul
b6Req4ersq/oVvU6sMbbpzny2NUQgtKnc2gARiE9ZmtjfZa3ZAOJEfcCgYEA6+vf
QKTCH5OkoCxxkRTPWTD5HAymch7afa4AWCBOBA/PnnjX5VgyEUa6RJS9hJuo0xPv
k0NsKT+avrxL8TVN0FhELYu3KR5rvBQll5fh3j3inRNlt9dQrLo/ZXou4wvDjMq7
b+rltcy5DW/LE7fnu9O+jBXpP5q8GhFUQLBkTOkCgYBu9ALpmXT+JLxRpBUMWB+6
nE40AZzzaPs52xIxOSKkWxOABSTJEbV6iJJOZuadWncYvwbGDz0izi4ddcll0X+A
g2QVbdDm2c5vWSV40snWoZlz3nrNdojSVkp+f0PSCoTlHrHwqiwdrQNg7FB4Yeid
VGDrfSc9ZwbqhpYG6GRcuQKBgHRvqHLGHgzpmHF5Hd/FzoQtmHTQhYGDWsB2lTLp
VrfIeeazq5JEkglWx+27H0IIdS0gBLjAqOw+uQfz9OMqNKpkNdnov9AqdG+R3g9m
pZJDJsnxXdl7BhUJoupZgDSMhMnC8PlOlFBSpu5Lw4WK2XMpET0gP0sl+jo/XT7E
D+ypAoGBAIfZ8zC7UDlfr/xidF+GUJlYoMabnhlbJfpnJ2A+oW/rBQgaCFHEIA52
Du0b2AVJZLACmGyaEZ/h+ZmKAJrJ7ptCE7Pi6oNm/wruDvuNOkCr3xYY2BhQdjMN
ubrCPqFykGH36E6rcYsLEd/5esjShtvHHgty/jGb/Yvztg6BvB+P
-----END RSA PRIVATE KEY-----
```

Figure 16-3a. *The private key*

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4QYm+aUBDY2hZ8RLLdmv
CtgeGjGhg0A5+9nITcAjFEWQ9ELTOA8iO+k8EO+ATJWMN5+/2xj1QZgvVpVx83MO
l03XopBWYZvCljftWaBsCX97NI7ltadjNnO5KnIboSV/YqMIBRzVCOFmtF756KlW
cBMj0pb/M5IcmcfTWHZAt7lNYuePtLDuVrWc/M3YALJycsSaicFH3ecp9SP7bhg3
VgMOueGllc3KFgqfyZitD6TJAYrGFx4CS/IjCo+/4odjNeGXL47KIDOGm3X7XnMH
tu86IqWLIM0m7xsmRR5FT/qBAjCf2/BrElofJPff2YkeZbFA9BtpjUWVrsK4vZKt
zwIDAQAB
-----END PUBLIC KEY-----
```

Figure 16-3b. *The public key*

In the two figures, note the difference between the length of the keys. Further, to use them in a C++ program, you will read them from the `.pem` files. First, you need to remove the extra messages from the files that are not part of the keys, namely the first line and the last line of the files. Make sure that there are no additional space characters left at the end of the keys, to not alter them.

Continuing the simulation for the cloud platform messaging, the encryption and the decryption are shown in Listing 16-1 and the output is given in Figure 16-4. Here, for demonstration purposes, we use a simple XOR-ing algorithm for both encryption and decryption. Make sure that `publicKey.pem` and `privateKey.pem` are in the same folder as the `.cpp` file containing the code from below.

Listing 16-1. Encryption and Decryption of the Messages

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
// the encryption scheme is a simple XOR-ing process
// XOR-ing is used for both encryption and decryption
// parameter "message" can be the plain message or the encrypted message,
according to user's needs
string xor_string(string message, string key)
{
    string out_message(message);
    unsigned int key_len(key.length()), message_len(message.length()), pos(0);
```

```
    for(unsigned int index = 0; index < message_len; index++)
    {
        out_message[index] = message[index] ^ key[pos];
        if(++pos == key_len){ pos = 0; }
    }
    return out_message;
}

int main()
{
    // read the message to be encrypted from the console
    string plain_text;
    cout<<"Enter the message: ";
    getline (cin, plain_text);

    // the public key is read from the .pem corresponding file
    string row1;
    string public_key = "";
    ifstream public_key_file ("publicKey.pem");
    if (public_key_file.is_open())
    {
        while (getline (public_key_file, row1) )
        {
            public_key += row1;
        }
        public_key_file.close();
    }
    // to check that the public key is read correctly, it is dispalyed on
       the console
    cout<<"Public key:"<<endl<<public_key<<endl<<endl;

    // the private key is read from the .pem corresponding file
    string row2;
    string private_key = "";
    ifstream private_key_file ("privateKey.pem");
```

```
    if (private_key_file.is_open())
    {
        while (getline (private_key_file, row2) )
        {
            private_key += row2;
        }
        private_key_file.close();
    }
    // to check that the public key is read correctly, it is dispalyed on
       the console
    cout<<"Private key:"<<endl<<private_key<<endl<<endl;

    // the encryption of the plain message is stored into encrypted_message
    string encrypted_text = xor_string(plain_text, public_key);
    cout << endl << "The encryption of the message is: " << endl <<
    encrypted_text << endl;

    // to decrypt the message, the receiver should proceed with some steps
    // 1. the receiver should xor his/her private key with his/her public key
    string xor_keys = xor_string(public_key, private_key);

    // 2. the receiver should xor the encrypted text with the result from
       the step 1
    string xor_result = xor_string(encrypted_text, xor_keys);

    // 3. the decryption is made by xor-ing the result from previous step
       with the private key
    string decrypted_message = xor_string(xor_result, private_key);

    cout << endl << "The decryption of the message is:  " << endl <<
    decrypted_message << endl;
    return 0;
}
```

Figure 16-4. *The output of Listing 16-1*

In Figure 16-4, note that the private and public keys don't contain the extra messages that were initially included in the `.pem` files.

Conclusion

This chapter covered the most important cryptographic primitives in cloud environments. At the end of this chapter, you should understand the cloud computing security issues and the advanced concepts and cryptographic primitives that can be applied to prevent these issues.

Cloud computing cryptography represents strong challenges and the huge amount of literature offers multiple theoretical frameworks that do not have real practical directions. This gives professionals and researchers strong research directions to develop new ideas for improving security in a cloud environment, excepting the standard security policies that are made available by the cloud solution providers.

References

[1] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos, "Security in cloud computing: Opportunities and challenges" in *Information Sciences*, Vol. 305 (pp. 357–383). 2015.

[2] OpenSSL. Available online: www.openssl.org/.

[3] OpenSSL download. Available online: https://sourceforge.net/projects/openssl/files/openssl-1.0.2d-fips-2.0.10/openssl-1.0.2d-fips-2.0.10.zip/download.

[4] Rishav Chatterjee, Sharmistha Roy, and Ug Scholar, *Cryptography in Cloud Computing: A Basic Approach to Ensure Security in Cloud*. 2017.

[5] N. Jaber and Mohamad Fadli Bin Zolkipli, "Use of cryptography in cloud computing" in *2013 IEEE International Conference on Control System, Computing and Engineering*, Mindeb, (pp. 179-184). doi: 10.1109/ICCSCE.2013.6719955. 2013.

[6] J.P. Kaur and R. Kaur, *Security Issues and Use of Cryptography in Cloud Computing*. 2014.

[7] Melissa Chase and Seny Kamara, "Structured Encryption and Controlled Disclosure" in *Advances in Cryptology-ASIACRYPT* (pp. 577-594). 10.1007/978-3-642-17373-8_33. 2010.

[8] M. Brenner, J. Wiebelitz, G.V. Voigt, and M. Smith, "Secret Program Execution in the Cloud Applying Homomorphic Encryption" in *Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies* (IEEE DEST 2011).

PART III

Pro Cryptanalysis

CHAPTER 17

Getting Started with Cryptanalysis

The third part of this book deals with cryptanalysis and its methods. As we mentioned in the beginning of the this book, *cryptanalysis* is the discipline that study the methods and ways of finding breaches within cryptographic algorithms and security systems. The final goal is to gain access to the real nature of the encrypted messages or cryptographic keys.

Cryptanalysis is a process that should be conducted by authorized persons, such as professionals (ethical hackers, information security officers, etc.). Any cryptanalysis activity outside of the legal framework is known as *hacking*, which covers personal and non-personal interests.

In this part, we will cover the most important methods and techniques for conducting cryptanalysis in general and in-depth. We will discuss the necessary knowledge and tools, such as software tools, methods, cryptanalysis types and algorithms, and penetration-testing platforms.

Conducting cryptanalysis can be a tricky and difficult task and many aspects must be taken into consideration before doing it. If you conduct the cryptanalysis as a *legal entity*, things become much easier. If the cryptanalysis is conducted by a *non-legal entity*, then you are dealing with a more complex process and hacking methods are involved, methods that will be covered later in our discussion. This being said, in both ways you need to get your hands dirty. The process of cryptanalysis is time-consuming and many obstacles and obstructions could occur for many reasons, such as system complexity, high size of the cryptographic key, hardware platform, access permissions, and so on.

Cryptanalysis is more exciting and challenging compared to cryptography. The knowledge that a cryptanalyst needs to have is very wide and complex. It covers several

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_17

complex fields that can be divided into three main categories: *informatics (computer science), computer engineering,* and *mathematics*. Let's specify the important disciplines for each of the categories as follows:

- Informatics (computer science)
  - Computer networks
  - Programming languages
  - Databases
  - Operating systems
- Computer engineering and hardware
  - FPGA (Field Programmable Gateway Array)
  - Programming languages (e.g. VHDL)
  - Development platforms (Xilinx, etc.)
- Mathematics
  - Number theory
  - Algebra
  - Combinatorics
  - Information theory
  - Probability theory
  - Statistical analysis
  - Elliptic curve mathematics
  - Discrete mathematics
  - Calculus
  - Lattices
  - Real analysis
  - Complex analysis
  - Fourier analysis

Third Part Structure

The purpose of the third part of this book is to provide the tools for implementing and providing the methods, algorithms, implementations of attacks, and designing and implementing a cryptanalysis strategy.

The third part structure is as follows:

- *Chapter 18:* The chapter will introduce a classification of cryptanalysis and techniques used in association with field of cryptanalysis. We will go through the theory of algorithm complexity, statistical-informational analysis, encoding in absence of perturbation, cryptanalysis of classic ciphers, cryptanalysis of block ciphers, and more.
- *Chapter 19:* The chapter will discuss linear and differential cryptanalysis. Their importance is quite crucial when cryptanalysis is performed.
- *Chapter 20:* The chapter will cover the integral cryptanalytic attack, which can be applied only for block ciphers that are based over substitution-permutation networks.
- *Chapter 21:* The chapter will study the behavior of software applications when they are exposed to different attacks and the source code is exploited.
- *Chapter 22:* This chapter will cover the most important techniques that can be used on text characterization. We will cover the chi-squared statistic; monogram, bigram, and trigram frequency counts; quadgram statistics as a fitness measure, and more.
- *Chapter 23:* We will cover in this chapter some case studies for implementing cryptanalysis methods.

Cryptanalysis Terms

In this section, we will introduce a list of cryptanalysis keywords and terms that are frequently used in the field. It is very important to get used to the terms before proceeding. This will help you have a clear image on the process and who interacts with what.

Table 17-1. *Cryptanalysis Terms*

Keyword/Term	Definition
Black hat hacker	A *black hat hacker* is a person who has a bad intention and breaks a computer system or network. His intention is to exploit any security vulnerability for financial gain; steal and destroy confidential and private data; shut down systems and websites; corrupt network communication, and so on.
Gray hat hacker	A *gray hat hacker* is a person, known as *cracker,* who exploits the security weak points of a computer system or software product with the goal of bringing those weaknesses to the owner's attention. Compared to a *black hat hacker,* a gray hat hacker will take action without any malicious intention. The *general goal* of a gray hat is to provide solutions and to improve the computer systems and security of the network.
White hat hacker/ethical hacker	A *white hat hacker* is an authorized person or certified hacker who is working for or employed by a government or organization with the goal of performing penetration tests and identifying loopholes within their systems.
Green hat hacker	A *green hat hacker* is an amateur person, but different from a *script kiddie.* Their purpose is to become a full-blown hacker.
Script kiddies	*Script kiddies* are the most dangerous hackers. A *script kiddie* is a person without many skills who uses scripts or downloads provided by other hackers. Their goal is to attack networks infrastructures and computer systems. They are looking to impress their community or friends.
Blue hat hacker	A *blue hat hacker* is similar to a script kiddie. They are beginners in the field of hacking. If someone dares to mock a script kiddie, then a blue hat hacker will get revenge. Blue hat hackers will get revenge on anyone who challenges them.
Red hat hacker	Known also as an eagle-eye hacker, their goal is to stop black hat hackers. The operation mode is different. They are ruthless when dealing with malware actions that come from black hat hackers. The attacks performed by red hat hackers are very aggressive.
Hacktivist	They are known as online activists. A hacktivist is a hacker who is part of a group of anonymous hackers who have the ability to gain unauthorized access to files stored within government computers and networks that serve social or political parties and groups.

(*continued*)

Table 17-1. (*continued*)

Keyword/Term	Definition
Malicious insider/ whistleblower	Such persons can be an employee of a company or government institution who is aware of illegal actions that take place within the institution. This could lead to a personal gain by blackmailing the institution.
State- or nation-sponsored hackers	This type of hacker is a person who is scheduled and assigned by a government with the goal of providing information security services and gaining access to confidential information from different countries. As an example, consider the malicious computer worm Stuxnet from 2010, which was designed and engineered to bring down the Iranian nuclear program. Another example is the United States 8th Air Force, which in 2009 became the US Cyber Command.

A Little Bit of Cryptanalysis History

A comprehensive history of cryptanalysis is very challenging so in this section we will cover some aspects and moments in time that influenced cryptanalysis as a separate field and how it evolved through different periods of history.

The history of cryptanalysis starts with Al-Kindi (801-873), the father of Arab philosophy. He discovered and developed a method based on the variations of the occurrence frequency of letters, a method that helped him analyze and exploit different ways of breaking ciphers (e.g. frequency analysis). The work of Al-Kindi was influenced by Al-Khalil's (717-786) work. Al-Khalil wrote the *Book of Cryptographic Messages*, which contained permutations and combinations for all possible Arabic words (both types of words, with and without vowels).

One of the best ways to learn the history of cryptanalysis and cryptography is to divide the subject into periods of time. It is very important to examine cryptanalysis history with respect for cryptography. Below, we provide a short classification of cryptanalysis history and focus on the most important achievements of each period.

- **600 B.C.**: *The Spartans* invent the scytale with the goal of sending secret messages during their fights. The device is composed of a leather strap and a piece of wooden stem. In order to decrypt the message, the wooden stem needs to be a specific size, the size used when the message was encrypted. If the receiver or malicious person doesn't have the same size wooden stem, the message can't be decrypted.

- **60 B.F.**: *Julius Caesar* sets the basis for the first substitution cipher, which encodes the message using shifting techniques for the characters using three spots: A will be D, B will be E, and so on. An implementation of this cipher can be seen in XXX.
- **1474**: *Cicco Simonetta* writes a manual for deciphering encryptions for Latin and Italian text.
- **1523**: *Blaise de Vigenère* introduces his encryption cipher, known as the Vigènere cipher.
- **1553**: *Giovan Battista Bellaso* creates the basis for the first cipher using an encryption key. The encryption key is characterized as a word that is agreed upon by the sender and the receiver.
- **1854**: *Charles Wheatstone* creates the Playfair Cipher. The cipher encrypts a specific set of letters instead of encrypting letter by letter. This raises the complexity of the cipher and in conclusion it becomes harder to crack.
- **1917**: *Edward Hebern* creates the first electro-mechanical machine in which the rotor from the machine is used for encryption operation. The encryption key is stored within a rotating disc. It has a table used for substitution, which is modified with every character that is typed.
- **1918**: *Arthur Scherbius* creates the Enigma machine. The first prototype is for commercial purposes. Compared to Edward Hebern's machine in which one rotor is used, the Enigma machine uses several rotors. The German Military Intelligence immediately adopts his invention for encoding their transmissions.
- **1932**: *Marian Rejewski* studies the Enigma machine and finds out how it operates. Starting in 1939, French and British Intelligence Services use the information provided by Poland, giving cryptographers such as Alan Turing the ability to crack the key, which changes on a daily basis. This is vital for the victory of Allies in World War II.

- **1945**: *Claude E. Shannon* publishes his work entitled *A Mathematical Theory of Cryptography*. This is the point when the classic cryptography period ends and modern cryptography begins.
- **End of 1970**: IBM creates a block cipher with the goal of protecting the data of the customers.
- **1973**: The United States adopts the block cipher and sets it as a national standard, called the DES (Data Encryption Standard).
- **1975**: Public key cryptography is introduced.
- **1976**: The Diffie-Hellamn key exchange is invented.
- **1982**: *Richard Feynman* introduces a theoretical model of a quantum computer.
- **1997**: The DES is cracked.
- **1994**: *Peter Shor* introduces an algorithm for quantum computers dedicated to integer factorization.
- **1998**: Quantum computing is introduced.
- **2000**: DES is officially replaced with the AES (Advanced Encryption Standard). AES won through an open competition.
- **2016**: IBM launches the IBM Q Experience with a five qubit quantum processor.
- **2017**: The appearance of Q# (Q Sharp) from Microsoft, a domain-specific programming language used for the implementation of quantum algorithms and cryptography applications.

This list can continue and be improved. We included the main events that contributed to the appearance of cryptanalysis as a concept, model, and framework.

Penetration Tools and Frameworks

In this section, we will cover several penetration tools and frameworks that can be used with success in the process of penetration testing, a process that is conducted by a certified professional.

We divided the tools into two categories: *Linux hacking distributions* and *penetration tools/frameworks*:

- **Linux hacking distributions**
  - **Kali Linux**: The most advanced platform for penetration testing. It has support for different devices and hardware platforms.
  - **BackBox**: A Linux distribution for penetration testing. It also includes security assessment.
  - **Parrot Security OS**: This distribution is quite new in this sphere. Its purpose and target is the cloud environment. It provides online anonymity and a strong encryption system.
  - **BlackArch**: A penetration testing platform and security research. It is built on top of Arch Linux.
  - **Bugtraq**: An impressive platform with forensic and penetration tools.
  - **DEFT Linux**: **D**igital **E**vidence & **F**orensics **T**oolkit (DEFT) is a very important distribution for computer forensics with the possibility of running as a live system.
  - **Samurai Web Testing Framework**: The framework and distro is a very powerful collection of tools that can be used in penetration testing on the Web. It's worth mentioning is that it comes as a virtual machine file, supported by VirtualBox and VMWare.
  - **Pentoo Linux**: Based on Gentoo, the distribution's intent is security and penetration testing. Available as live.
  - **CAINE**: **C**omputer **A**ided **In**vestigative **E**nvironment, it is a powerful distribution that offers a serious set of system forensics modules and analysis.
  - **Network Security Toolkit**: One of the favourite tools and distributions is Network Security Toolkit, a live ISO build on Fedora. It contains a very important set of open source network security tools. It provides a professional web user interface for network and system administration, network monitoring tools, and analysis.

- **Fedora Security Spin**: A professional distro for security audit and tests. It can be used by various types of professionals, from industry to academia.
- ***ArchStrike***: Also known as ArchAssault, it is a distro built on Arch Linux for professionals in the field of security and penetration testers.
- **Cyborg Hawk**: Contains more than 750 tools for security professional and performing penetration tests.
- **Matriux**: The distribution is quite promising and it can be used for penetration tests, ethical hacking, forensic investigations, vulnerability analysis, and much more.
- **Weakerth4n**: Not well-known in the field of hacking or cryptanalaysis, Weakerth4n offfers an interesting approach to penetration tests and it is built using Debian (Squeeze).

- **Penetration tools/frameworks (Windows and Linux platforms)**
  - **Wireshark**: A very well-known packet sniffer. Provides a powerful set of tools for network packages and protocol analysis.
  - **Metasploit**: One of the most important frameworks for pentesting, the framework will develop and execute vulnerabilities exploitation.
  - **Nmap**: **N**etwork **M**apper is a very powerful network discovery and security auditing tool for security professionals. Its goal is to exploit their targets. For each port you are scanning, you can see what OS is installed, what services are running, what firewall is installed and used, etc.

Conclusion

In this section, we discussed cryptanalysis in general and we covered the basic foundation of cryptanalysis, its tools, and working methods. At the end of this chapter, you should be able to

- Understand the mission and goal of cryptanalysis
- Understand the main events during the course of history and how the appearance of difference ciphers and algorithms influenced the cryptanalysis discipline
- Define common terms and the differences between different types of hackers
- Understand the hacking and penetration platform distributions
- Understand the most important frameworks and penetration tools that can be used independently, according to the OS platform

References

[1] F. Cohen, "A short history of cryptography." New World Encyclopedia, 1990. (2007). Retrieved May 4, 2009, from www.all.net/books/ip/Chap2-1.html.

[2] Cryptography. Retrieved May 4, 2009, from www.newworldencyclopedia.org/entry/Cryptography.

[3] M. Pawlan, "Cryptography: the ancient art of secret messages." February, 1998. Retrieved May 4, 2009, from www.pawlan.com/Monica/crypto/.

[4] J. Rubin, "Vigenere Cipher." 2008. Retrieved May 4, 2009, from www.juliantrubin.com/encyclopedia/mathematics/vigenere_cipher.html.

[5] K. Taylor, "Number theory 1." July 31, 2002. Retrieved May 4, 2009, from http://math.usask.ca/encryption/lessons/lesson00/page1.html.

[6] M. Whitman and H. Mattord, *Principles of information security*. [University of Phoenix Custom Edition e-text]. Canada, Thomson Learning, Inc. 2005. Retrieved May 4, 2009, from University of Phoenix, rEsource, CMGT/432.

[7] Simon Singh, *The Code Book. The Secret History of Codes and Code-Breaking*. 1999

[8] A. Ibraham, "Al-Kindi: The origins of cryptology: The Arab contributions" in *Crypto logia*, vol.16, no 2 (pp. 97-126). April, 1992. www.history.mcs.st-andrews.ac.uk/history/Mathematicians/Al-Kindi.html.

[9] Abu Yusuf Yaqub ibn Ishaq al-Sabbah Al-Kindi. www.trincoll.edu/depts/phil/philo/phils/muslim/kindi.html.

[10] Philosophers: Yaqub Ibn Ishaq al-Kindi Kennedy-Day, K. al-Kindi, Abu Yusuf Ya'qub ibn Ishaq (d. c.866–73). www.muslimphilosophy.com/ip/kin.html.

[11] Ahmad Fouad Al-Ehwany, "Al-Kindi" in *A History of Muslim Philosophy*, Volume 1 (pp. 421-434). New Delhi: Low Price Publications. 1961.

[12] Ismail R. Al-Faruqi and Lois Lamya Al-Faruqi, *Cultural Atlas of Islam* (pp. 305-306). New York: Macmillan Publishing Company. 1986.

[13] Encyclopaedia Britannica (pp. 352). Encyclopaedia Britannica, Inc. Chicago: William Benton. 1969.

[14] J.J. O'Connor and E.F. Robertson, "Abu Yusuf Yaqub ibn Ishaq al-Sabbah Al-Kindi." 1999.

CHAPTER 18

Cryptanalysis Attacks and Techniques

In this chapter, we will cover the most important and useful cryptanalytic and cryptanalysis standards, validation methods, classifications, and cryptanalysis attacks.

The cryptanalysis discipline is very wide and writing about it could take up thousands of pages. In the following sections, we will go through all the necessary elements that are necessary for developers to use in their daily activities.

Standards

It is very important to understand the importance of standards when you are conducting cryptanalysis attacks for business purposes only, with the goal of testing the security within an organization.

The main institutes and organizations that provide high standards for cryptography and cryptanalysis methods, frameworks, and algorithms are

- **IEFT Public-Key Infrastructure (X.509)**: The organization deals with the standardization of protocols used on the Internet that are based on public key systems.
- **National Institute of Standards and Technologies (NIST)**: The institute deals with the elaboration of FIPS standards for the US government.
- **American National Standards Institute (ANSI)**: Its purpose is to administer the standards from the private sector.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_18

- **Internet Engineering Task Force (IEFT)**: An international community of networks, operators, traders of services, and researchers who deal with the evolution of the Internet architecture.
- **Institute of Electrical and Electronical Engineering (IEEE)**: Its objective is to elaborate on theories and advanced techniques from different fields, such as electronics, computer sciences, and informatics.
- **International Organization for Standardization (ISO)**: It represents a non-governmental organism of more than 100 countries. Its main purpose is to promote the developing of standardization in order to facilitate the international exchange of services.

FIPS 140-2, FIPS 140-3, and ISO 15408

ISO 15408 is the evaluation of IT security and it is used in the international community as a reference system. The standard defines a set of rules and requirements from the IT field with the goal of validating the security of the product and cryptographic systems.

FIPS 140-2/140-3 is a set of guidelines that need to be followed in order to fulfill a specific set of technical requirements that are exposed on four levels.

When you develop a specification or criteria for a certain application or cryptographic module, you must take into consideration FIPS 140-2/FIPS 140-3 and ISO 15408. Products that are developed with respect for the mentioned standards need to be tested in order to get a validation and to confirm that the criteria was followed and respected properly.

Validation of Cryptographic Systems

If the business requires cryptanalysis and cryptography operations to be implemented within the software and communication systems, then cryptographic and cryptanalysis services are required. These services are authorized by certification organizations and include functionalities such as digital signature generation and verification, encryption and decryption, key generation, key distribution, key exchange, etc.

Figure 18-1 depicts a general model for testing security based on cryptographic and cryptanalysis modules.

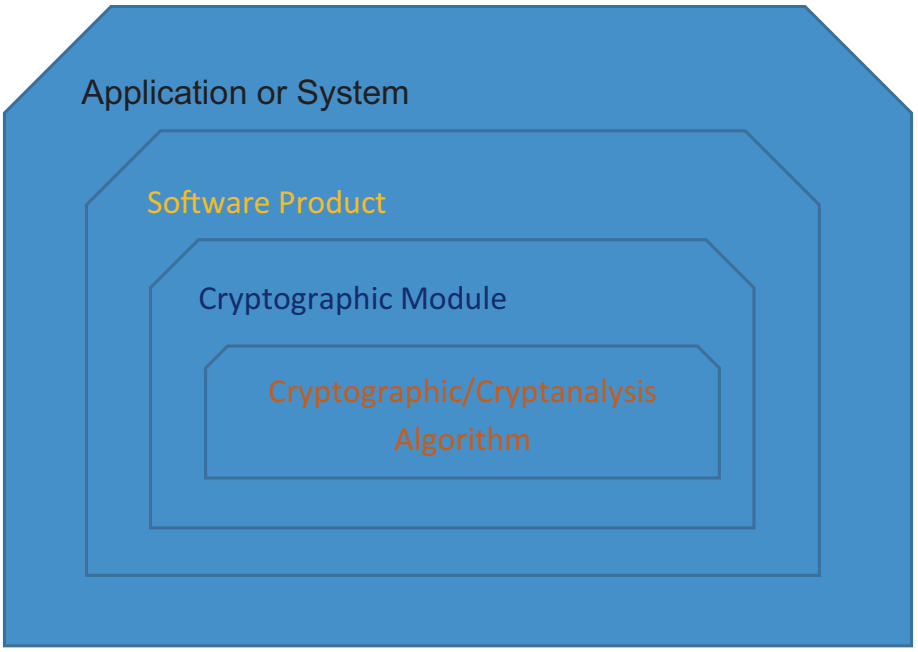


Figure 18-1. *Verification and testing framework*

For a proper testing and verification process, it is necessary and to have two minimum steps, a *cryptographic/cryptanalysis algorithm* and a *cryptographic module*. For example, if you are developing a cryptographic product or a desktop or web software application, it is necessary for the company/institute/developer to perform the tests and to send them to CMVP[1] (Cryptographic Module Validation Programme) in order to be tested with respect for FIPS 140-2[2] and FIPS 140-3[3].

A *cryptographic module* represents a specialized combination of software and hardware processes. The main advantages of using validated cryptographic and cryptanalysis modules are the following:

- Making sure the modules have satisfied the necessary requirements
- Making sure that the authorized and technical personnel is informed and instructed within a standard that is commonly agreed upon and was tested.

[1]CMVP, https://csrc.nist.gov/projects/cryptographic-module-validation-program
[2]FIPS 140-2, https://csrc.nist.gov/publications/detail/fips/140/2/final
[3]FIPS 140-3, https://csrc.nist.gov/publications/detail/fips/140/3/final

- Making sure that the final user (end user) is aware of the fact that the cryptographic module was verified and tested in accordance with some well-defined security requirements
- A high level of reliability for security, which need to be fulfilled with the goal of developing similar and specific applications

The security requirements of FIPS 140-2 contain 11 metrics for designing and implementing the cryptographic module. For each cryptographic module validated, the following requirements need to be fulfilled. During the validation process, the cryptographic modules receive a mark from 1 to 4, which is proportional to the security level guaranteed.

The cryptographic modules, once validated, contain information such as the name of the manufacturer, address, name of the module, version of the module, type of module (software or hardware), validation date, validation level, and module description.

Cryptanalysis Operations

Designing a cryptographic system has to be done following these simple *principles*:

- The opponent should not be underestimated.
- The security of a cryptographic system can be evaluated by a cryptanalyst.
- Before the evaluation of the cryptographic system is performed, knowledge about the adversaries is taken into consideration for the evaluated cryptosystem.
- The secret of the cryptographic system has to rely on the key.
- The process of the cryptographic system evaluation has to take into consideration all the elements within the system, such as key distribution, cryptographic content, etc.

According to the father of information theory, Claude Elwood Shannon[4], the following must be taken into consideration when the evaluation of the cryptosystem is performed:

- One of the winnings of the cryptanalyst is gained once the message is decrypted with success

[4]Claude Elwood Shannon, www.itsoc.org/about/shannon

- The key length and complexity
- The level of complexity of performing a encryption-decryption process
- The size of the encrypted text in accordance with the text size
- The way of error propagation

The basic operations for having a solution for each cryptogram are as follows:

- Finding and determining the language used
- Determining the cryptographic system
- Reconstructing a specific key for a cryptographic system or a partial or complete reconstructing for a stream cryptographic system
- Reconstruction of such system or establishing the complete plaintext

Classification of Cryptanalytics Attacks

This section covers the types of attacks on cipher algorithms, cryptographic keys, and authentication protocols on the protocols, systems, and hardware attacks.

Attacks on Cipher Algorithms

Table 18-1. *Attacks on Ciphering Algorithms*

Types of Attacks on Ciphering Algorithms	
Attack Title	**Attack Description**
Known plaintext attack	The cryptanalyst has an encrypted text and his correspondent has the plaintext. The goal of this attack is for the cryptanalyst to separate the encryption key from the information.
Chosen text attack	The cryptanalyst has the possibility to indicate the plaintext that will be encrypted. The cryptanalyst will try to separate the information of the text from the encryption key, having the possibility to obtain through specific methods the encryption algorithm or the key.

(*continued*)

Table 18-1. (*continued*)

Types of Attacks on Ciphering Algorithms	
Attack Title	**Attack Description**
Cipher-cipher text attack	The cryptanalyst holds a plaintext and his correspondent the same text, which is encrypted with two or more different keys.
Divide-et-imperia attack	The cryptanalyst has the chance to realize a series of correlations between different inputs and outputs of the algorithm with the goal of separating different inputs in the algorithm, which makes them break the problem into two or more problems that are easy to solve.
Linear syndrome attack	The cryptanalysis method consists of designing and creating a linear equation system specific to the pseudorandom generator and verifying the equation system with the encrypted text, obtaining in this way the plaintext with a high probability.
Consistency linear attack	The cryptanalytic method consists of the elaboration of a linear equation system specific to the pseudorandom generator starting from an equivalent cryptographic key and verifying the system by the pseudorandom generator with the probability, which goes to 1, obtaining in this way the plaintext with a high probability.
Stochastic attack	Known also as a forecasting attack, the attack is possible if the output of the generator is autocorrelated, the cryptanalyst managing to obtain as input data the output of the pseudorandom generator and the encrypted text. In this way the clear text is obtained.
Informational linear attack	Known also as a linear complexity attack, the attack is possible if there is any chance to equalize the generator with a Fibonacci algorithm and additionally if the linear complexity of the generator is low. With this type of attack, it is possible to build a similar algorithm and a similar cryptographic key.
Virus attack	This attack is possible if the encryption algorithm is implemented and is run on a PC that is vulnerable and unprotected.

Attacks on Cryptographic Keys

The most frequent attacks that occur on cryptographic keys are listed in Table 18-2.

Table 18-2. *Attacks on Cryptographic Keys*

Types of Attacks on the Keys	
Attack Title	**Attack Description**
Brute force attack	The attack consists in the exhaustive verification of keys and passwords, and it is possible if the encryption key size is small and the encryption key space is small.
Intelligent brute force attack	The level of key randomness of the encryption key is small (the entropy is small) and allows finding the password, which is similar to words from the utilized language.
Backtracking attack	The attack is based on the implementation of the method of a backtracking type, which consists of the existence of conditions for continuing the search in the desired direction.
Greedy attack	The attack provides the optimal local key, which cannot be the same as the optimal global key.
Dictionary attack	The attack consists of searching for passwords or keys and is done using a dictionary.
Hybrid dictionary attack	This attack is done by modifying words from the dictionary, initializing the brute force attack with the help of the words from the dictionary.
Viruses attack	This attack is possible if the keys are stored on an unprotected PC.
Password hash attack/ cryptographic key	This attack takes place if the hash of the password is short or wrongly elaborated.
Substitution attack	The original key is substituted by a third party and replaced in the entire network. This can be done with the help of viruses.
Storing encryption key	If this is done in a wrong way (together with the encryption data) in plaintext without any physical protection measures or cryptographic software or hardware, it can lead to an attack on the encrypted message.
Storing of old encryption keys	This attack will compromise old documents that are encrypted.

(*continued*)

Table 18-2. (*continued*)

Types of Attacks on the Keys	
Attack Title	**Attack Description**
Key compromise	If the symmetric key is compromised, only the documents that are assigned with that key will be compromised. If the public key is compromised, which can be found stored on different servers, the attacker can be substituted with the legal owner of the data, resulting in a negative impact within the network.
Master keys	Represents different phases in the cryptographic system.
Key lifetime	It is an essential component that excludes the possibility of a successful attack that was undetected.

Attacks on Authentication Protocols

The authentication protocols are exposed to different types of attacks. Table 18-3 lists the most important ones, which are frequently used. Note that the authentication protocol of a system is very important and vital. Once corrupted, vital information could be exposed and attackers could gain a lot of benefits: personal, financial, and so on.

Table 18-3. *Attacks on Authentication Protocols*

Types of Attacks on Authentication Protocols	
Attack Title	**Attack Description**
Attack on the public key	The attack takes place for the signature within the protocol. This is available only for systems with public keys.
Attack on the symmetric algorithm	The attack takes place on the signature within the authentication protocol. This is available only if a symmetric key is used.
Passive attack	The attacker intercepts and monitors the communication on the channel without making any kind of intervention.
Attack using a third person	The communication of the two partners within a communication channel is intercepted actively by a third party.
The fail-stop signature	A cryptographic protocol in which the sender can bring evidence if his signature was forged or not.

Conclusion

The chapter covered the most important and useful cryptanalytic and cryptanalysis guidelines and methods. You can now manage the standards with the goal of testing and verifying the implementation of the cryptographic and cryptanalytic algorithms and methods. As a summary, you learned about

- Cryptanalysis attack classification
- Cryptanalysis operations
- Standards FIPS 140-2 and FIPS 140-3
- Standard 15408
- Validation of cryptographic systems

References

[1] Adrian Atanasiu, *Matematici in criptografie*. US Publishing House, Editor: Universul Stiintific, ISBN: 978-973-1944-48-7. 2015.

[2] Adrian Atanasiu, *Securitatea Informatiei vol 1 (Criptografie)*. InfoData Publishing House, ISBN: 978-973-1803-18-0. 2007.

[3] Adrian Atanasiu, *Securitatea Informatiei vol 2 (Protocoale de securitate)*. InfoData Publishing House, ISBN: 978-973-1803-18-0. 2009.

[4] S.J. Knapskog, "Formal specification and verification of secure communication protocols," 58–73.

[5] K. Koyama, "Direct demonstration of the power to break public-key cryptosystems," 14–21.

[6] P.J. Lee, "Secure user access control for public networks," 46–57.

[7] R. Lidl and W.B. Muller, "A note on strong Fibonacci pseudoprimes," 311–317.

[8] Menezes, S. Vanstone, “The implementation of elliptic curve cryptosystems,” 2–13.

[9] M.J. Mihaljević and J.D. Golić, “A fast iterative algorithm for a shift register initial state reconstruction given the noisy output sequence,” 165–175.

CHAPTER 19

Linear and Differential Cryptanalysis

In this chapter, we will discuss two important types of cryptanalysis: *linear* and *differential cryptanalysis.* To explain how to merge theoretical concepts with the practical, in the beginning we will go through a set of basic concepts and advanced techniques on how these two types of cryptanalysis can be implemented by professionals.

Despite the fact that some of the differential and linear mechanisms are outdated, there is plenty of room to find new challenges that can be exploited in order to obtain new results. The research literature about linear and differential cryptanalysis provides a significant amount of theoretical approaches and mechanisms, but only few of the theories could be applied in practice to develop professional solutions for differential and linear cryptanalysis attacks.

The difference between theoretical and applied cryptanalysis is significantly huge. The results that were published over the last 12 years, such as algorithms, methods, game theory aspects, and so on led researchers and professionals on different paths. Most of them were whimsical chimeras (complex mathematical systems without real applicability) and fancy algorithms; others were applicable with success in practice.

Conducting research in cryptanalysis and increasing its potential value for being applied in practice and for different scenarios requires time, experience, and a continuous cross-collaboration between theoreticians and practitioners, without existing an isolation between these two types of categories. Their importance is crucial in the field of cryptanalysis, providing the necessary tools and mechanisms to construct cryptanalysis attack schemes for block and stream ciphers.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_19

Differential Cryptanalysis

Differential cryptanalysis was implemented by E. Biham and A. Shamir in the early 90s. The objective of differential cryptanalysis is to check whether the cryptogram traces some locations from the key with a probability greater than others. The checking process can be carried out with any order with grade 1. Actually, the test represents a complicated approximation of order 2 of a test cycle.

With differential cryptanalysis we will expose the weak points of the cryptography algorithm. The following example of differential cryptanalysis is illustrated for *stream cryptography algorithms.* In the following pseudocode, we will illustrate the algorithm as follows:

```
INPUT:    The key is chosen as K = (k1, ..., kn) with ki ∈ {0, 1}
OUTPUT:   The weak points of the cryptography
          algorithm together with the resistance decision for
          differential cryptanalysis
```

1. $\alpha \leftarrow$rejection rate value
2. Choose n sets of keys with perturbation property set, starting from the key K.

$$for\ i = 1\ to\ n\ do\ K^{(i)} = \left(\delta_{1i} \oplus k_1, \ldots, \delta_{ni} \oplus k_n\right):$$

$$\delta_{1i} = \begin{cases} 1, if\ j \neq i, \\ 0, if\ j = i. \end{cases},$$

for i,j=1,…,n. In the form from above the i[th] key is obtained from the base key by changing the bit from i[th] position.

3. **Constructing the cryptograms**: The first step is to build $n + 1$ cryptograms using the basic key, perturbed keys, and a clear text M. We denote the obtained cryptograms as $C^{(i)}$, $i = 1, \ldots, n + 1$. As plaintext M we can choose for text 0 – everywhere.
4. **Constructing the correlation matrix**: Here we build the matrix $(n + 1) \times (n + 1)$ for the corellation values C:

$$c_{ij} = corellation\left(cryptogram\ i, cryptogram\ j\right).$$

Correlation c_{ij} denotes the value of the statistical test applied to the sequence (*cryptogram i* $\oplus$ *cryptogram j*). The matrix C is represented as a symmetrical matrix having 1 on the main diagonal.

5. **The computational process for the significant value**: It counts the values of significant correlation that are situated above the main diagonal. A value is called *significant* if

$$c_{i,j} \notin \left[u_{\frac{\alpha}{2}}; u_{1-\frac{\alpha}{2}} \right].$$

Consider T the number of significant values that represent the number of rejects of the correlation test.

6. **Decision and result interpretation**: If

$$\frac{T - \alpha \cdot \frac{n(n+1)}{2}}{\sqrt{\alpha(1-\alpha) \cdot \frac{n(n+1)}{2}}} \notin \left[u_{\frac{\alpha}{2}}; u_{1-\frac{\alpha}{2}} \right],$$

once computed, we can decide the non-resistance to differential cryptanalysis ($u_{\frac{\alpha}{2}}$ and $u_{1-\frac{\alpha}{2}}$) represents the quantiles of the normal distribution of order $\frac{\alpha}{2}$ and $1-\frac{\alpha}{2}$ and fixes the (i,j) elements with $n \geq i > j \geq 1$, for which c_{ij} is significant. These elements represent weak points for the algorithms. Otherwise we would not be able to mention anything about the resistance to this type of attack.

Listing 19-1 shows the implementation in C++ of the above pseudocode and the output shown in Figure 19-1 is self-explanatory and quite intuitive. The source code is divided into seven steps:

- Generating a differential structure as a matrix
- Computing the differences
- Computing the intermediate values that are generated using the differentials
- Generating the known key pairs
- Computing and looking for a good pair of keys

- Using brute force to reduce the space of the keys
- Computing and displaying the key(s) pair(s)

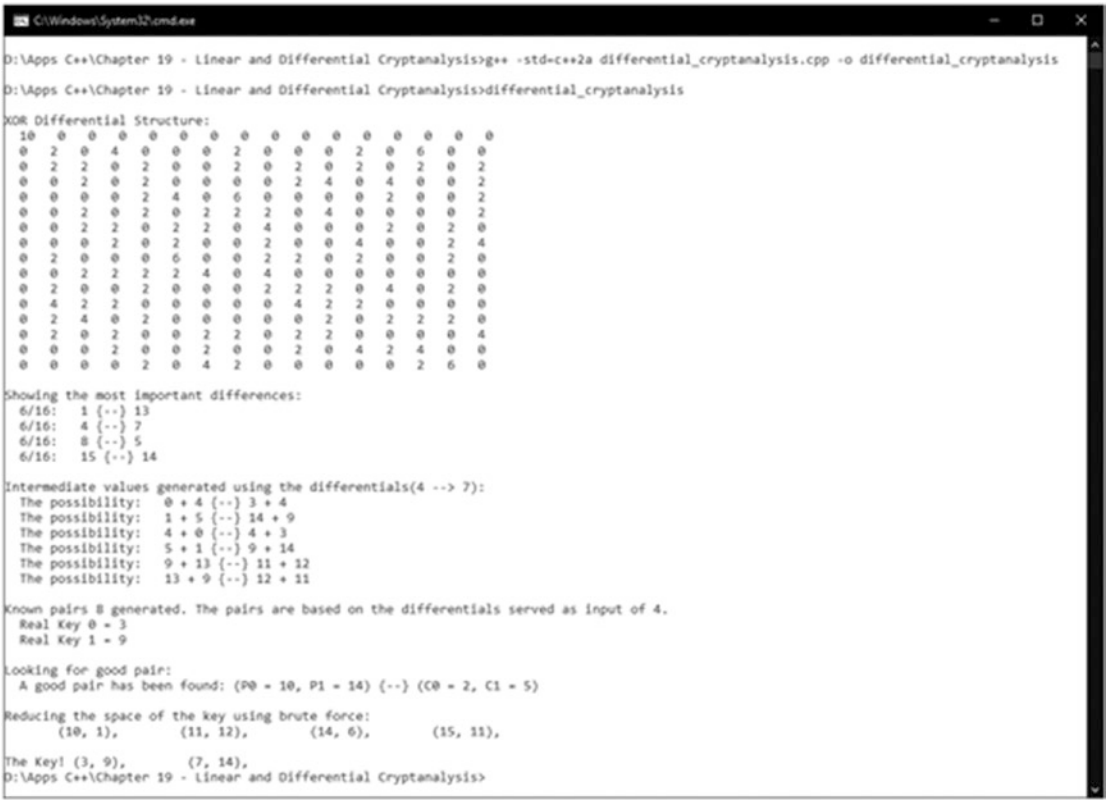


Figure 19-1. *Differential cyptanalysis example*

Listing 19-1. Implementation of the Differential Cryptanalysis Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//** variables
int theSBox[16] = {3, 14, 1, 10, 4, 9, 5, 6, 8, 11, 15, 2, 13, 12, 0, 7};
int characters[16][16];
int known_plaintext_0[10000];
int known_plaintext_1[10000];
int known_ciphertext_0[10000];
```

```
int known_ciphertext_1[10000];
int good_plaintext_0, good_plaintext_1, good_ciphertext_0, good_ciphertext_1;
int pairs_of_numbers;
int characters_dataSet_0[16];
int characters_data_max = 0;
int round_function(int integer_input, int cryptoKey)
{
    return theSBox[cryptoKey ^ integer_input];
}

int encryption(int integer_input, int key_0, int key_1)
{
    int x_value_0 = round_function(integer_input, key_0);
    return x_value_0 ^ key_1;
}

void find_differences()
{
    printf("\nXOR Differential Structure:\n");

    int a, b, c, f;    //c, d, e, f

    for(a = 0; a < 16; a++)
    {
        for(b = 0; b < 16; b++)
        {
            characters[a ^ b][theSBox[a] ^ theSBox[b]]++;
        }
    }

    for(a = 0; a < 16; a++)
    {
        for(b = 0; b < 16; b++)
            printf("  %x ", characters[a][b]);
        printf("\n");
    }
```

```
    printf("\nShowing the most important differences:\n");

    for(a = 0; a < 16; a++)
     {
        for(b = 0; b < 16; b++)
         {
            if (characters[a][b] == 6)
              {
                  printf("  6/16:   %i {--} %i\n", a, b);
               }
         }
    }
}

void generate_characters_data(int input_differences,
                              int output_differences)
{
    printf("\nIntermediate values generated using the
           differentials(%i --> %i):\n", input_differences,
           output_differences);

            characters_data_max = 0;
            int f;
            for(f = 0; f < 16; f++)
            {
                int computations = f ^ input_differences;

                if ((theSBox[f] ^ theSBox[computations]) ==
                     output_differences)
                {
                 printf("  The possibility:   %i + %i {--}
                              %i + %i\n", f, computations,
                              theSBox[f], theSBox[computations]);
                characters_dataSet_0[characters_data_max] = f;
                    characters_data_max++;
                }
            }
}
```

```
void generate_pairs(int input_differences)
{
    printf("\nKnown pairs %i generated. The pairs are based on
           the differentials served as input of %i.\n",
           pairs_of_numbers, input_differences);

     //** generate substitution keys
    int real_key_0 = rand() % 16;
    int real_key_1 = rand() % 16;

    printf("  Real Key 0 = %i\n", real_key_0);
    printf("  Real Key 1 = %i\n", real_key_1);

    int c;

    //** using XOR, pairs for plaintexts and
    //** ciphertexts are generated
    for(c = 0; c < pairs_of_numbers; c++)
    {
        known_plaintext_0[c] = rand() % 16;
        known_plaintext_1[c] = known_plaintext_0[c] ^
                               input_differences;
        known_ciphertext_0[c] = encryption(known_plaintext_0[c], real_key_0,
                    real_key_1);
        known_ciphertext_1[c] = encryption(known_plaintext_1[c], real_key_0,
                    real_key_1);
    }
}

void identifying_good_pair(int output_differences)
{
    printf("\nLooking for good pair:\n");
    int c;

     //** test if the pair of ciphertexts meet
    //** the characteristics
```

```
    for(c = 0; c < pairs_of_numbers; c++)
     {
        if ((known_ciphertext_0[c] ^ known_ciphertext_1[c]) == output_
        differences)
        {
            good_ciphertext_0 = known_ciphertext_0[c];
            good_ciphertext_1 = known_ciphertext_1[c];
            good_plaintext_0 = known_plaintext_0[c];
            good_plaintext_1 = known_plaintext_1[c];
               printf("  A good pair has been found: (P0 =
                        %i, P1 = %i) {--} (C0 = %i, C1 = %i)\n",
                        good_plaintext_0, good_plaintext_1,
                        good_ciphertext_0, good_ciphertext_1);
            return;
        }
     }
     printf("There was no good pair found!\n");
}

int key_testing(int key_test_0, int key_test_1)
{
    int c;
    int crap = 0;
    for(c = 0; c < pairs_of_numbers; c++)
    {
        if ((encryption(known_plaintext_0[c], key_test_0,
                  key_test_1) != known_ciphertext_0[c]) ||
                  (encryption(known_plaintext_1[c], key_test_0,
                  key_test_1) != known_ciphertext_1[c]))
        {
            crap = 1;
            break;
        }
    }
```

```
    if (crap == 0)
        return 1;
    else
        return 0;
}

void brute_cracking()
{
    printf("\nReducing the space of the key using brute force:\n");

    int f;

    //** based on the characteristics, we will
    //** test each of the possible value
    for(f = 0; f < characters_data_max; f++)
    {
        int key_test_0 = characters_dataSet_0[f] ^
                                 good_plaintext_0;
        int key_test_1 = theSBox[characters_dataSet_0[f]] ^
                                 good_ciphertext_0 ;

        if (key_testing(key_test_0, key_test_1) == 1)
            printf("\n\nThe Key! (%i, %i), ", key_test_0,
                                              key_test_1);
        else
            printf(" (%i, %i), ", key_test_0, key_test_1);
    }
}
int main()
{
     //** generate random value once the program is run
    srand(time(NULL));

     //** look in the s-boxes for good differences
    find_differences();
```

```
    //** define number of known pairs
     pairs_of_numbers = 8;

    //** look for inputs that meet specific characteristics
     generate_characters_data(4, 7);

     //** let's generate pairs of chosen-plaintext
     generate_pairs(4);

     //** pick a pair which meet the characteristic
     identifying_good_pair(7);

     //** find the key
    brute_cracking();
}
```

Linear Cryptanalysis

Linear cryptanalysis was developed as a theoretical framework for the DES (data encryption system) and was implemented in 1993. Linear cryptanalysis is commonly used inside block ciphers and is a very good starting point for designing and executing complex attacks.

Linear cryptanalysis is defined as a linear relationship that is set between the key, the plaintext structure, and the ciphertext structure. The plaintext is structured and represented as characters or bits. It is required to have a structure of a chain of operations characterized by exclusive-or, as we describe here,

$$A_{i_1} \oplus A_{i_2} \ldots \oplus A_{i_u} \oplus B_{j_1} \oplus B_{j_2} \oplus \ldots \oplus B_{j_v} = Key_{k_1} \oplus Key_{k_2} \oplus \ldots Key_{k_w},$$

where $\bigoplus$ represents the XOR operation as a binary operation, A_i represents the bit from i^{th} position of input the structure $A = [A_1, A_2, \ldots]$, B_j represents the bit from j^{th} position of the output structure $B = [B_1, B_2, \ldots]$, and Key_k represents the k^{th} bit of the key $Key = [Key_1, Key_2, \ldots]$.

Performing Linear Cryptanalysis

Usually, in the most important cases, performing a linear cryptanalysis starts from the idea that we are aware and we acknowledge the encryption algorithm except the private key. Executing a linear cryptanalysis against a block cipher is represented as a framework described here:

- The first step is based on identifying the linear approximation for non-linear components. The goal is to characterize the encryption algorithm (as an example, S-Boxes).
- The next step is to compute a combination between linear approximations of substitution boxes that also includes the operations that are executed against the encryption algorithm. Professionals should focus on the linear approximation due to the fact that it represents a special function which contains and deals with the clear text and cipher text bits together with the ones from the private key.
- Computing and designing the linear approximation should be done as a guideline with respect for the cryptographic keys that are used for the first time. The guideline proves its power and will help professionals save important computational resources for all the possible values of the cryptographic keys. Based on using multiple linear approximations, we will have a very powerful process of computation with the goal of eliminating the key numbers which are necessary for trying.

The next sections provide extra details of the components that are taken into consideration when conducting a linear cryptanalysis attack. Without having a clear image of the theory mechanisms, the practical concepts, and how to put them into practice, it will be a very difficult task to launch a real attack.

S-Boxes

Using S-Boxes the non-linearity is introduced together with its operations, *exclusive-or* and *bit-shift*, which are found within the linear representation as well.

The *scope of an S-Box* is to design and create a map between the incoming binary sequences with a specific and requested output. In the end, we will have the non-linearity provided that will build and render the affine approximation that was computed with the help of the linear cryptanalysis applied. Table 19-1 shows an example of an S-Box and how the mapping works. The S-Box uses the 1^{st} and 4^{th} bit to find the column and the middle bits, 3^{rd} and 4^{th}. Using this approach, the row is determined in such way that input 1110 will be 0101.

Table 19-1. *S-Box Example*

	11	10	01	00
00	0000	0001	0010	0011
01	1000	1001	1111	1011
10	1100	1101	1110	1010
11	0100	0101	0010	0111

In Table 19-2, the mapping operation is demonstrated between examples of bits as input and bits as output.

Table 19-2. *The Mapping Between Input and Output*

The input (J)	The output (Q)
0000	0011
0001	0010
0010	1011
0011	1111
0100	1010
0101	1110
0110	0111
0111	0010
1000	0001
1001	0000
1010	1001
1011	1000
1100	1101
1101	1100
1110	0101
1111	0100

Linear Approximation of S-Box

We start from the idea that we want to approximate the structure of the substitution box presented above. Based on that information, we have the precision, which is quite high, of the various linear approximations. We include 256 such linear approximations having the following form:

$$d_1 J_1 \oplus d_2 I_2 \oplus d_3 J_3 \oplus a_4 I_4 = g_1 Q_1 \oplus g_2 Q_2 \oplus g_3 Q_3 \oplus g_4 Q_4$$

where J_1 and Q_i represent the i^{th} bit characterized to input (J) and ouput (Q) with respect for d_i and g_1, which are 0 or 1. As an example, let's use the following linear approximation $J_2 = Q_1 \bigoplus Q_4$ and being given by $d = 0100_2$ and $g = 1001_2$.

Concatenation of Linear Approximations

It's time to form, design, and project the linear approximation for the whole system. To achieve this, we need two things:

- First, we need to have computed already the linear approximation for each component that forms the system.
- Second, to do the combination. For this, we simply *sum* by using exclusive-or the entire set of equations in different combinations. In this way we get a single equation, which eliminates the intermediate variables.

Assembling Two Variables

Let's consider B_1 and B_2, two random binary variables. The linear relationship between them is $B_1 \bigoplus B_2 = 0$. Next, we denote the probability $B_1 = 0$ by being noted with l and the probability $B_2 = 0$ by being noted with m. Based on the two random independent variables, we have

$$P(B_1 = a, B_2 = b) = \begin{cases} l \cdot m & for\ a = 0, b = 0 \\ l \cdot (1-m) & for\ a = 0, b = 1 \\ (1-l) \cdot q & for\ a = 1, b = 0 \\ (1-l) \cdot (1-m) & for\ a = 1, b = 1 \end{cases}$$

Moving forward, we can show the following:

$$P\left(B_1 \oplus B_2 = 0\right) =$$

$$= P\left(B_1 = B_2\right)$$

$$= P\left(B_1 = 0, B_2 = 0\right) + P\left(B_1 = 1, B_2 = 1\right)$$

$$= l \cdot m + \left(1 - l\right)\left(1 - m\right)$$

The next step is represented by computing the bias for $B_1 \bigoplus B_2 = 0$; it will be given by $\zeta_1 \cdot \zeta_2$.

This being said, it is time to do the implementation in C++ (see Listing 19-2) for the linear cryptanalysis (see Figure 19-2) and to show how the above concepts can be used in practice.

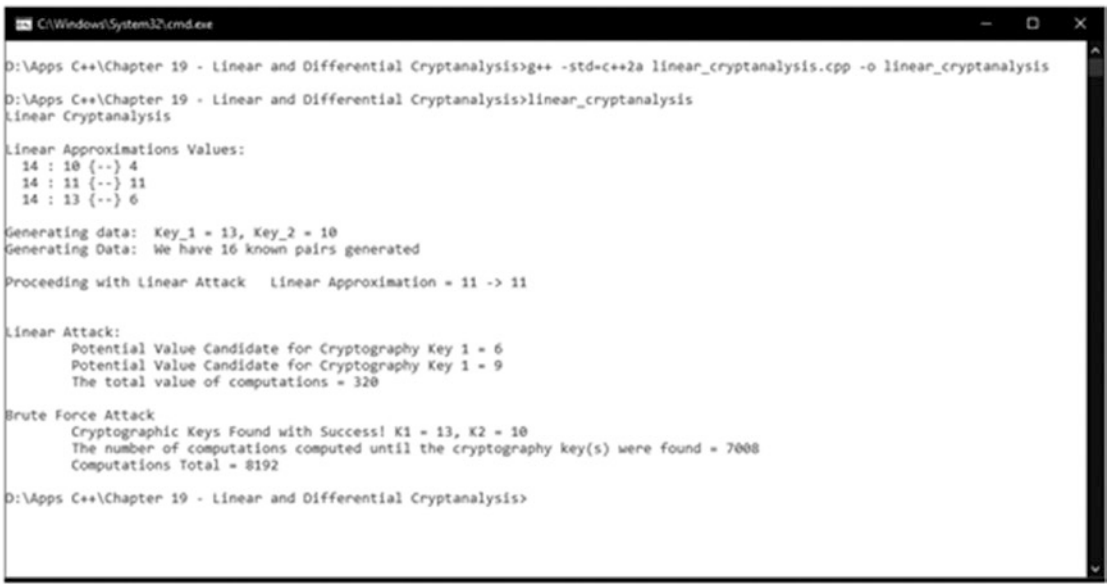


Figure 19-2. *Linear cryptanalysis output simulation program*

Listing 19-2. Linear Cryptanalysis Simulation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int sBox_content[16] = {9, 11, 12, 4, 10, 1, 2, 6, 13, 7, 3,
                        8, 15, 14, 0, 5};

int sBox_content_revision[16] = {14, 5, 6, 10, 3, 15, 7, 9,
                                 11, 0, 4, 1, 2, 8, 13, 12};

int approximation_array[16][16];
int known_plaintext[500];
int known_ciphertext[500];
int numbers_known = 0;

int using_mask(int inputValue, int mask_value)
{
    int value = inputValue & mask_value;
    int total = 0;

    while(value > 0)
    {
        int temporary = value % 2;
        value /= 2;

        if (temporary == 1)
          {
            total = total ^ 1;
          }
    }
    return total;
}

void looking_for_approximation()
{
    int a, b, c;
```

```
    //** the output of mask value
    for(a = 1; a < 16; a++)
    {
          //** the input of mask value
          for(b = 1; b < 16; b++)
          {
            //** input
            for(c = 0; c < 16; c++)
               {
                if (using_mask(c, b) ==
                     using_mask(sBox_content[c], a))
                    {
                    approximation_array[b][a]++;
                    }
               }
          }
     }
}

void display_approximation()
{
    printf("Linear Approximations Values: \n");
    int a, b, c;
    for(a = 1; a < 16; a++)
        for(b = 1; b < 16; b++)
            if (approximation_array[a][b] == 14)
                printf("  %i : %i {--} %i\n",
                      approximation_array[a][b], a, b);

    printf("\n");
}

int round_function(int inputValue, int substitution_key)
{
    return sBox_content[inputValue ^ substitution_key];
}
```

```
void filling_known_numbers()
{
    int substitution_key_1 = rand() % 16;
    int substitution_key_2 = rand() % 16;

    printf("Generating data:  Key_1 = %i, Key_2 = %i\n",
                        substitution_key_1, substitution_key_2);

    int total = 0;

    int c;
    for(c = 0; c < numbers_known; c++)
    {
        known_plaintext[c] = rand() % 16;
        known_ciphertext[c] =
                  round_function(round_function(known_plaintext[c],
                  substitution_key_1), substitution_key_2);
    }

    printf("Generating Data:  We have %i known pairs
                   generated\n\n", numbers_known);

}

void keys_testing(int key_1, int key_2)          //testKeys
{
    int c;
    for(c = 0; c < numbers_known; c++)
     {
        if (round_function(round_function(known_plaintext[c],
                    key_1), key_2) != known_ciphertext[c])
          {
            break;
          }
     }
    printf("# ");
}
```

```
int main()
{
    printf("Linear Cryptanalysis\n\n");

    srand(time(NULL));

    looking_for_approximation();
    display_approximation();

    int input_approximation = 11;
    int output_approximation = 11;

     //** how many numbers we known
    numbers_known = 16;
    filling_known_numbers();

    int cryptographic_key_score[16];
    int reaching_threshold = 0;

    printf("Proceeding with Linear Attack");
    printf("\tLinear Approximation = %i -> %i\n",
              input_approximation, output_approximation);

    printf("\n\n");

    int b, h;
    for(b = 0; b < 16; b++)
    {
        cryptographic_key_score[b] = 0;
        for(h = 0; h < numbers_known; h++)
        {
            reaching_threshold++;
            int middle_round = round_function(known_plaintext[h], b);

            if ((using_mask(middle_round, input_approximation)
                        == using_mask(known_ciphertext[h],
                        output_approximation)))
                cryptographic_key_score[b]++;
```

```
        else
            cryptographic_key_score[b]--;

    }
}

int maximum_score_value = 0;

for(b = 0; b < 16; b++)
{
    int score_value = cryptographic_key_score[b] *
                      cryptographic_key_score[b];
    if (score_value > maximum_score_value)
               maximum_score_value = score_value;
}

int good_cryptographic_keys[16];

for(h = 0; h < 16; h++)
    good_cryptographic_keys[h] = -1;

  h = 0;
  printf("Linear Attack:\n");
  for(b = 0; b < 16; b++)
 {
    if ((cryptographic_key_score[b] *
         cryptographic_key_score[b]) == maximum_score_value)
    {
        good_cryptographic_keys[h] = b;
           printf("\tPotential Value Candidate for
                  Cryptography Key 1 = %i\n",
                  good_cryptographic_keys[h]);
        h++;
    }
 }

  int guessing_cryptographic_key_1;
  int guessing_cryptographic_key_2;
```

```
    for(h = 0; h < 16; h++)
    {
        if (good_cryptographic_keys[h] != -1)
        {
               int cryptography_key_test_1 =
                       round_function(known_plaintext[0],
                       good_cryptographic_keys[h]) ^
                       sBox_content_revision[
                              known_ciphertext[0]];

               int tested = 0;
               int e;
               int bad = 0;
               for(e = 0;e < numbers_known; e++)
               {
                    reaching_threshold += 2;
                    int testOut = round_function(round_function
                                  (known_plaintext[e],
                                  good_cryptographic_keys[h]),
                                  cryptography_key_test_1);
                    if (testOut != known_ciphertext[e])
                         bad = 1;
               }
               if (bad == 0)
               {
                    printf("\tFound Keys! K1 = %i, K2 =
                           %i\n", good_cryptographic_keys[h],
                                cryptography_key_test_1);
                    guessing_cryptographic_key_1 =
                                      good_cryptographic_keys[h];
                    guessing_cryptographic_key_2 =
                                      cryptography_key_test_1;
                    printf("\tComputations Until Key Found =
                                      %i\n", reaching_threshold);
```

```
            }
        }
    }
    printf("\tThe total value of computations = %i\n\n",
                                        reaching_threshold);
    reaching_threshold = 0;

     printf("Brute Force Attack\n");
    for(h = 0; h < 16; h++)
    {
          for(b = 0; b < 16; b++)
          {
               int e;
               int bad = 0;
               for(e = 0;e < numbers_known; e++)
               {
                    reaching_threshold += 2;
                    int testOut = round_function(round_function(
                                   known_plaintext[e], h), b);
                    if (testOut != known_ciphertext[e])
                         bad = 1;
               }
               if (bad == 0)
               {
                    printf("\tCryptographic Keys Found with
                           Success! K1 = %i, K2 = %i\n", h, b);
                    printf("\tThe number of computations
                              computed until the cryptography
                              key(s) were found = %i\n",
                              reaching_threshold);
               }
          }
    }
    printf("\tComputations Total = %i\n", reaching_threshold);
}
```

Conclusion

The chapter discussed differential and linear cryptanalysis attacks and how these kinds of attacks can be designed and implemented in real practice. We introduced the theoretical background elements and main foundations, which must be understood before you design such cryptanalysis attacks.

At the end of this chapter, you can now

- Identify theoretically the main components on which a cryptanalyst should focus
- Understand how vulnerable those components are and how they can be exploited
- Implement linear and differential cryptanalysis attacks

References

[1] Joan Daemen, Lars Knudsen, and Vincent Rijmen "The Block Cipher Square. Fast Software Encryption (FSE)" in *Volume 1267 of Lecture Notes in Computer Science* (pp. 149–165). Haifa, Israel: Springer-Verlag. CiteSeerX 10.1.1.55.6109. 1997.

[2] H. Heys, "A Tutorial on Linear and Differential Cryptanalysis" in *Cryptologia*, vol. XXVI, no. 3 (pp. 189-221). 2002.

[3] M. Matsui, "Linear Cryptanalysis Method for DES Cipher" in *Advances in Cryptology - EUROCRYPT '93* (pp. 386-397). Springer-Verlag. 1994.

[4] E. Biham, "On Matsui's linear cryptanalysis" in (ed. A. De Santis) *Lecture Notes in Computer Science*, vol. 950 (pp. 341–355). Springer-Verlag; Berlin. 1995.

[5] A. Biryukov, C. De Cannière, and M. Quisquater, "On Multiple Linear Approximations" in (ed. M. Franklin) *Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152* (pp. 1-22). Springer-Verlag. 2004.

[6] L. Keliher, H. Meijer, and S.E. Tavares (2001). "New method for upper bounding the maximum average linear hull probability for SPNs" in (ed. B. Pfitzmann) *LNCS*, vol. 2045 (pp. 420–436). Springer-Verlag; Berlin. 2001.

[7] L.R. Knudsen and J.E. Mathiassen, "A chosen-plaintext linear attack on DES" in (ed. B. Schneier) *Lecture Notes in Computer Science*, vol. 1978 (pp. 262-272). Springer-Verlag; Berlin. 2001.

[8] M. Matsui and A. Yamagishi, "A new method for known plaintext attack of FEAL cipher" in (ed. R.A. Rueppel) *Lecture Notes in Computer Science*, vol. 658 (pp. 81-91). Springer-Verlag; Berlin. 1993.

[9] M. Matsui, "The first experimental cryptanalysis of the data encryption standard" in (ed. Y.G. Desmedt) *Lecture Notes in Computer Science*, vol. 839 (pp. 1-11). Springer-Verlag; Berlin. 1994.

CHAPTER 20

Integral Cryptanalysis

Integral cryptanalysis is a cryptanalytic technique that is designed for block ciphers constructed on networks of substitution-permutation. Since the integral cryptanalysis attack can be launched against the Square block cipher [1], it is also known as the Square attack. It was designed by Lars Knudsen.

An exposed point of the block cipher is the network of substitution-permutation. When the networks can be discovered (intuitively), then the exploitation of vulnerabilities of the block cipher has a high negative impact over the entire cryptosystem. Other exposed points of the block ciphers are the key itself and the table involved in the permutation of the key. When a false key is similar (or identical) to the correct one, then the system can be broken.

In the next section, we present the formal basis regarding block ciphers, which can be used in an implementation. Further, we present the elements required to initiate an integral cryptanalysis attack, for example, building Feistel networks and generating a permutation table for a cryptographic key. Once you have a clear understanding of these two phases, it becomes clear how integral cryptanalysis must be conducted.

Basic Notions

To implement and design an integral cryptanalytic attack, is very important to have the formal elements before starting to implement the attack. Moving further, let's take a look at the following concepts as the main starting point. They are used for designing and implementing such an attack for educational purposes only.

Take into consideration $(G, +)$ as a a finite abelian group with the order k. The product group $G^n = G \times \ldots \times G$ is a group of elements with the structure $v = (v_1, \ldots, v_n)$, where $v_i \in G$. The addition within G^n is defined as component-wise, therefore, we have $u + v = w$ holds for $u, v, w \in G^n$ when $u_i + v_i = w_i$ for all i.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_20

Let's denote B as the set with multiple vectors. We define the integral over B. This integral represents the sum of all vectors S. The integral is defined as $\int S = \sum_{v \in B} v$, and the addition operation is defined in terms of the group operation for G^n.

When the integral cryptanalytic is designed, it's important to know the number of words in the plain text and in the encrypted text. In the example from this chapter, this number is denoted with n. Another important number to know is the number of clear texts and the encrypted texts, denoted with m. In general, $m = k$ (i.e. $k = |G|$), the vectors $v \in B$ denotes the plain text and the encrypted text, and $G = GF(2^B)$ or $G = Z/kZ$.

Going further into the attack, it is based on the fact that one of the involved entities will make a prediction for the values placed in the integrals after a particular number of rounds of encryption. Keeping this in mind, three cases can be distinguished: (1) when the words have the same length (e.g. i), (2) when the words have different lengths, and (3) the sum of a particular value that is predicted in advance.

Further, consider $B \subseteq G^n$ as described above and a fixed index i. The following three cases can be distinguished:

(1) $v_i = c$, for all $v \in B$

(2) $\{v_i : v \in B\} = G$

(3) $\sum_{v \in B} v_i = c'$

where $c, c' \in G$ are some values known and fixed in advance.

The example that we present further is a common situation in which $m = k$, the number of vectors from B is the same as the number of elements in the considered group. From Lagrange's theorem, it results that if all words, a general word placed on the i^{th} position, have the same length, then it is intuitive that the i^{th} word from the integral will have the value of the neutral element from G.

The following two theorems are necessary and they represent a must for any practical developer who wants to translate into practice an integral cryptanalysis.

Theorem 20-1 [1, Theorem 1, p. 114]**.** Let's consider $(G, +)$ a *finite abelian additive group*. The subgroup of elements of order 1 or 2 is denoted as $L = \{g \in G : g + g = 0\}$. We will consider writing $s(G)$ as being the sum $\sum_{g \in G} g$ of all the elements found within G. Next, we will consider $s(G) = \sum_{h \in H} H$. More, it is very important to understand the following analogy: $s(G) \in H$, i.e. $s(G) + s(G) = 0$.

According to Theorem 20-1, for $G = GF(2^B)$ there is the value $s(G) = 0$ and for Z/mZ there is the value $s(Z/mZ) = m/2$ in the situation where m is an even value or it is 0. The following theorem represents the multiplicative case for written groups (see Theorem 20-2).

Theorem 20-2 [1, Theorem 2, p. 114]. Let's consider $(G, *)$ a *finite abelian multiplicative group*. The subgroup of elements of order 1 or 2 is denoted as $H = \{g \in G : g * g = 1\}$. We consider writing $p(G)$ as being the product $\prod_{g \in G} g$ of all the elements of G. Next, we consider $p(G) = \prod_{h \in H} h$. More, it is very important to understand the following analogy: $p(G) \in H$, i.e. $p(G) * p(G) = 1$.

As an example, if we have $G = (Z/pZ)^*$ where p is prime, $p(G)$ will be -1, $p(G) = -1$. This is proved using Wilson's theorem.

Practical Approach

In this section, we will implement an integral cryptanalysis attack that can be applied in practice using the C++ programming language.

The following approach presents a basic implementation of an integral cryptanalysis in C++, giving the chance to override the size of the current block (`sboxValue` with `revision_sbox_value`). In the provided implementation, we use repeating sequences. The goal of the repeating sequences is to create a weakness to show how it can be exploited and launch the attack.

In the source code from Listing 20-1, we demonstrate the integral cryptanalytic attack, providing necessary comments on building and designing principles. In Listing 20-1, you can see how the integral cryptanalysis attack is implemented. As soon as you have the main kernel of the program, to proceed further with the success of the attack, it is vital to "figure out" how the attack is designed and afterwards to create a copy of it or something that is similar to the original one.

To use the following example, you must run from the command prompt the following command (see Figure 20-1):

```
g++ -std=c++2a integral.cpp -o integral
```

Listing 20-1. The Main Program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//** VARIABLES
//** the sBox structure
int sboxValue[16] = {9, 11, 12, 4, 10, 1, 2, 6, 13, 7, 3, 8, 15, 14, 0, 5};
//** the revision sBox value structure
int revision_sbox_value[16] = {14, 5, 6, 10, 3, 15, 7, 9, 11, 0, 4, 1, 2,
8, 13, 12};
//** an approximation array
int approximation_array_structure[16][16];
//** the maximum score which will need to be compared with the "score"
variable
int maximum_score = 0;
//** the total value
int total_value = 0;
//** control counters, 1 and 2
int counter1;
int counter2;
//** generating known plaintext
int known_plaintext[500];
//** generating known ciphertext
int known_ciphertext[500];
//** number of generated known pairs
int known_numbers = 0;
//** approximation values, for input and output
int approximation_for_input = 11;
int approximation_for_output = 11;
//** score value for the cryptography key
int cryptographyKey_score_value[16];
//** the threshold level. used for identitying the step until we reached
for finding the cryptography keys
int threshold_level = 0;
```

```
//** good reliable cryptography key
int reliable_crypto_keys[16];
//** the values for the guessed cryptography key
int guessing_crypto_key1, guessing_crypto_key2;
//** testing control variable for cryptography key
int cryptoKey1_testing;

//** apply the mask
int applying_the_mask(int theValue, int mask_value)
{
    int internal_value = theValue & mask_value;
    int total_value = 0;

    while(internal_value > 0)
    {
        int temporary = internal_value % 2;
        internal_value /= 2;

        if (temporary == 1)
            total_value = total_value ^ 1;
    }
    return total_value;
}

//** detecting the approximation
void detect_the_approximation()
{
     int a, b, c;

     //** parsing for the output mask. 16 represents
     //** the size of the s-Box
    for(a = 1; a < 16; a++)
     {
        //** parsing for the input mask
        for(b = 1; b < 16; b++)
        {
            //** parsing the input
```

```
            for(c = 0; c < 16; c++)
            {
                if (applying_the_mask(c, b) ==
                    applying_the_mask(sboxValue[c], a))
                    approximation_array_structure[b][a]++;
            }
        }
    }
}

//** show and display the approximation
void display_the_approximation()
{
    printf("Integral Values Approximations: \n");
    int a, b, c;
    for(a = 1; a < 16; a++)
     {
        for(b = 1; b < 16; b++)
          {
            if (approximation_array_structure[a][b] == 14)
                printf("  %i : %i {<-->} %i\n",
                   approximation_array_structure[a][b], a, b);
          }
     }

    printf("\n");
}

//** round function for s
int rounding_procedure(int dataInput, int substitution_key)
{
    return sboxValue[dataInput ^ substitution_key];
}
```

```
//** filling the knowings possible key values
void filling_up_the_knowings()
{

    int substitution_key_1 = rand() % 16;
    int substitution_key_2 = rand() % 16;

    printf("Generating Data:  Key 1 = %i, Key 2 = %i\n",
                substitution_key_1, substitution_key_2);

    //** parse each known value and for each known plaintext
    //** and ciphertext compute the proper values
    for(counter1 = 0; counter1 < known_numbers; counter1++)
    {
        known_plaintext[counter1] = rand() % 16;
        known_ciphertext[counter1] =
            rounding_procedure(rounding_procedure(
            known_plaintext[counter1], substitution_key_1),
            substitution_key_2);
    }

    printf("Generating Data:  We have generated %i known
                      pairs\n\n", known_numbers);
}

//** verify and test the cryptography keys
void testing_the_keys(int key_1, int key_2)
{
    for(counter2 = 0; counter2 < known_numbers; counter2++)
        if
                  (rounding_procedure(rounding_procedure(
                   known_plaintext[counter2], key_1), key_2) !=
                   known_ciphertext[counter2])
            break;

    printf("* ");
}
```

```
int main()
{
    printf("Testing Program for Integral Cryptanalysis\n\n");

    srand(time(NULL));

    detect_the_approximation();
    display_the_approximation();

    known_numbers = 16;
    filling_up_the_knowings();

    printf("Integral Cryptanalysis Attack -->  Based on Linear
    Approximation = %i {<-->} %i\n", approximation_for_input,
    approximation_for_output);

    int c, d;
    for(c = 0; c < 16; c++)
    {
        cryptographyKey_score_value[c] = 0;
        for(d = 0; d < known_numbers; d++)
        {
            threshold_level++;
            int midRound =
                 rounding_procedure(known_plaintext[d], c);

            if ((applying_the_mask(midRound,
                                   approximation_for_input) ==
                        applying_the_mask(known_ciphertext[d],
                        approximation_for_output)))
                cryptographyKey_score_value[c]++;
            else
                cryptographyKey_score_value[c]--;
        }
    }
```

```
for(c = 0; c < 16; c++)
{
    int score = cryptographyKey_score_value[c] *
                cryptographyKey_score_value[c];
    if (score > maximum_score) maximum_score = score;
}

for(d = 0; d < 16; d++)
 {
    reliable_crypto_keys[d] = -1;
 }

d = 0;

for(c = 0; c < 16; c++)
 {
    if ((cryptographyKey_score_value[c] *
         cryptographyKey_score_value[c]) == maximum_score)
    {
        reliable_crypto_keys[d] = c;
        printf("Integral Cryptanalysis Attack -->
              Candidate for K1 = %i\n",
              reliable_crypto_keys[d]);
        d++;
    }
 }

for(d = 0; d < 16; d++)
{
    if (reliable_crypto_keys[d] != -1)
    {
            cryptoKey1_testing =
                    rounding_procedure(known_plaintext[0],
                    reliable_crypto_keys[d]) ^

                    revision_sbox_value[known_ciphertext[0]];
```

```
                int tested = 0;
                int e;
                int bad = 0;
                for(e = 0;e < known_numbers; e++)
                {
                    threshold_level += 2;
                    int testOut =
                        rounding_procedure(rounding_procedure(
                        known_plaintext[e],
                        reliable_crypto_keys[d]),
                        cryptoKey1_testing);
                    if (testOut != known_ciphertext[e])
                        bad = 1;
                }
                if (bad == 0)
                {
                    printf("Integral Cryptanalysis Attack -->
                           We have found the cryptography keys! Crypto Key
                           1 = %i, Crypto Key 2 = %i\n", reliable_crypto_
                           keys[d], cryptoKey1_testing);
                    guessing_crypto_key1 =
                                       reliable_crypto_keys[d];
                    guessing_crypto_key2 = cryptoKey1_testing;
                    printf("Integral Cryptanalysis Attack -->
                          Number of computations for reaching the
                          cryptography key = %i\n", threshold_level);

                }

        }
    }
    printf("Integral Cryptanalysis Attack --> Computations
                          Total = %i\n\n", threshold_level);

    threshold_level = 0;
```

```
    for(d = 0; d < 16; d++)
    {
            for(c = 0; c < 16; c++)
            {

                int e;
                int bad = 0;
                for(e = 0;e < known_numbers; e++)
                {
                    threshold_level += 2;
                    int testOut =
                        rounding_procedure(rounding_procedure(
                        known_plaintext[e], d), c);
                    if (testOut != known_ciphertext[e])
                        bad = 1;
                }
                if (bad == 0)
                {
                    printf("Brute Force -->  We have found the
                        cryptography keys! Crypto Key 1 = %i,
                        Crypto Key 2 = %i\n", d, c);
                    printf("Brute Force -->  Number of
                        computations for reaching the
                        cryptography key = %i\n",
                        threshold_level);
                }
            }
    }

    printf("Brute Force -->  Total computations number =
                                %i\n", threshold_level);

}
```

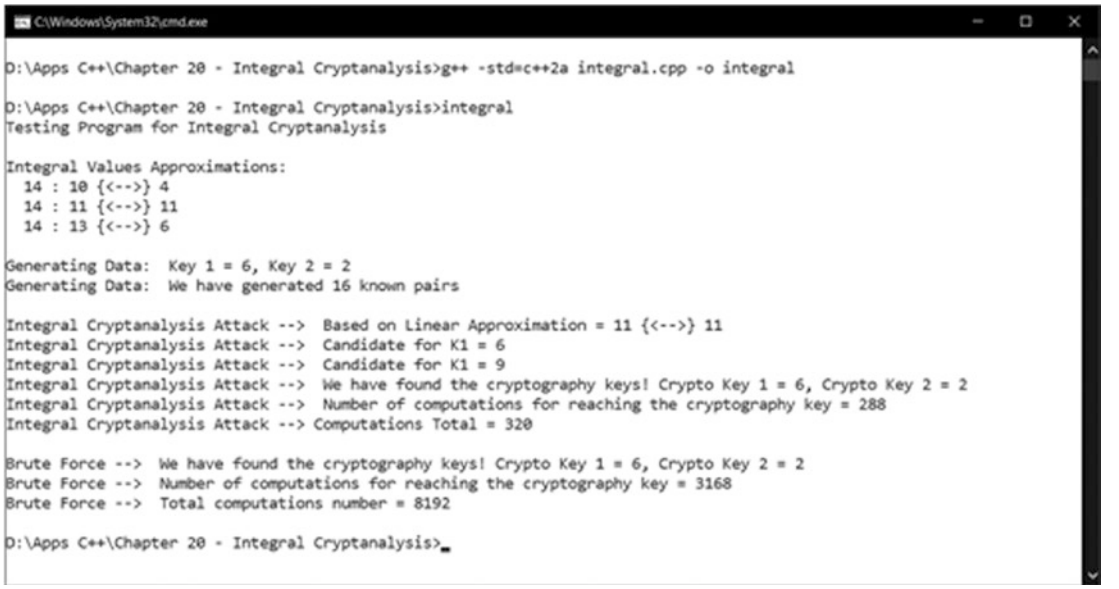


Figure 20-1. *Integral cryptanalysis attack*

Conclusion

The chapter covered integral cryptanalysis and how such an attack can be designed and implemented. The chapter covered a way of building a block cipher cryptosystem together with the vulnerable points with the goal of illustrating how to use an integral cryptanalysis attack in practice.

Now that you've reached the end of the chapter, you can

- Design and implement a simple integral cryptanalysis attack.
- Understand the vulnerable points of this kind of attack and generate the permutation tables with the goal of permutating the key.
- Use permutation tables and work with them over the keys.

Reference

[1] Joan Daemen, Lars Knudsen, and Vincent Rijmen, "The Block Cipher Square," Fast Software Encryption (FSE) 1997, *Volume 1267 of Lecture Notes in Computer Science*. Haifa, Israel; Springer-Verlag. pp. 149–165. CiteSeerX 10.1.1.55.6109. 1997.

CHAPTER 21

Brute Force and Buffer Overflow Attacks

This chapter covers two important attacks, the *buffer overflow* attack and the *brute force attack*, which are frequently employed against C++ applications and programs.

A special category of attackers will use applications/programs or devices to launch brute force or buffer overflow attacks. Their methods evaluate different combinations of words for confirmation forms. In some cases, the attackers attempt to corrupt web applications through a scanning process for sessions IDs, for example. The attacker's goals include stealing data, corrupting destination machines with malware, and asking for a specific amount of money. Some of the attackers perform brute force attacks physically as a personal choice. Today, most brute force and buffer overflow attacks are performed by bots.

In order to protect organizations and businesses from these kinds of attacks, take into consideration the following recommendations:

- Don't use data or information that could be found online.
- Use as many characters as possible.
- Use combinations of letters, numbers, and special characters (e.g. symbols).
- Don't use common patterns (e.g. qwerty).
- Use different passwords for each user account.
- Change the password at frequent intervals (e.g. every two months).
- Use and generate long and strong passwords. If you don't have an idea for one, use a password generator (e.g. key generation from KeyPass).

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_21

- Implement multifactor authentication [1].
- Use biometrics if possible [2].

Brute Force Attack

A brute-force attack represents a complex attack in which the attacker will submit many passwords or passphrases with the goal of guessing the correct one. Each password or passphrase is checked one-by-one by the attacker until the correct one is found. Also, the attacker may guess the key. The key is generally created from the password by using a function for key derivation. This process is known as an *exhaustive key search.*

There are many types of brute-force attacks:

- **Attacks based on rainbow tables**: A rainbow table is a predefined and precomputed table. The goal is to reverse the process of the cryptographic hash capacities.
- **Attacks based on reversing brute-force attacks**: The attack is based on using a general password or a specific set of passwords against multiple usernames.
- **Credential attacks**: The attack uses sets of passwords-usernames against a variety of websites.
- **Hybrid brute-force attacks**: The attack is used to figure out what password variety is used to succeed. After, it proceeds with a general process for dealing with checking different varieties.

We will illustrate this type of attack with the following examples to show how the attack can be used and deployed in real-life situations (algorithms). The examples provided are

- **Brute-force attack on a Caesar cipher**: The example (see Figure 21-1 and Listing 21-1) is based on a Caesar cipher. The example is chosen as the first case of showing a brute-force attack due to its simplicity.
- **String generation for a brute-force attack**: The example (see Figure 21-2 and Listing 21-2) shows how basic string generation can be done with the goal of creating complex lists and dictionaries that can be used during brute-force attacks.

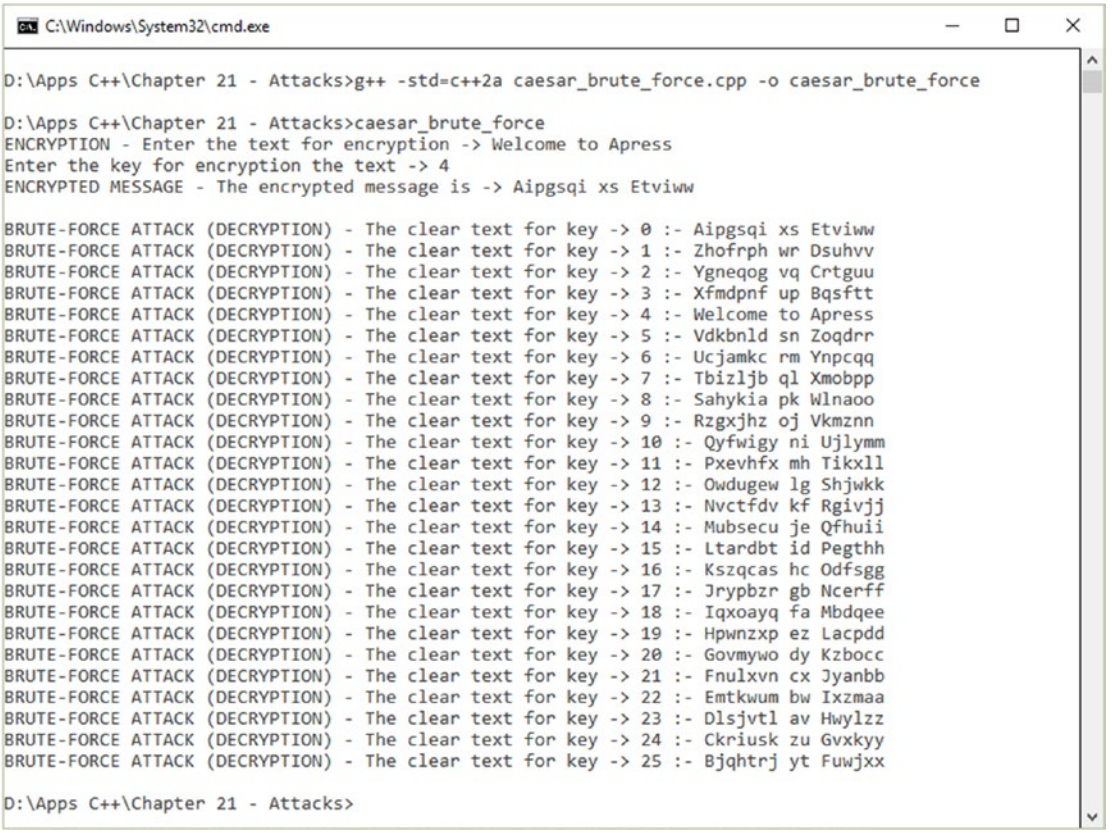


Figure 21-1. *Running the brute-force attack*

Listing 21-1. Brute Force Attack on a Caesar Cipher

```
#include<iostream>
using namespace std;

// the function will be used to encrypt the plaintext
// string msg - the message
// int keytValue - the key
string encrypt(string msg,int keyValue)
{
     // variable used to hold the cipher value of the plaintext
    string cipher="";
```

```
     // parse the string
    for(int i=0;i<msg.length();i++)
    {
        // verify if the character is upper case
        if(isupper(msg[i]))
           // add to the cipher the character plus the key and subtract
              ASCII 65 value ('A').
           // the value obtained do modulo 26 (english alphabet letters)
              and add ASCII value 65 back.
           cipher += (msg[i] + keyValue - 65)%26 + 65;

          // verify if the character is lower case
          else if(islower(msg[i]))
               //** the same as above. ASCII value 97 ('a')
               cipher += (msg[i] + keyValue - 97)%26 + 97;
          else
               cipher += msg[i];
    }
    return cipher;
}

// The decryption will be done using the brute force attack by
// checking all possible keys
// string encMessage - the encrypted message
void decrypt(string encMessage)
{
     // the variable for storing the plaintext
    string plaintext;

     // we will try for each key and we will do the decryption
    for(int keyTry=0;keyTry<26;keyTry++)
    {
        plaintext = "";
          // parse accordingly based on the message length
```

```
        for(int i=0;i<encMessage.length();i++)
        {
               // check if the character is upper case
            if(isupper(encMessage[i]))
            {
                if((encMessage[i] - keyTry - 65)<0)
                    plaintext += 91 + (encMessage[i] - keyTry - 65);
                else
                    plaintext += (encMessage[i] - keyTry - 65)%26 + 65;
            }
               // check if the character is lower case
            else if(islower(encMessage[i]))
            {
                if((encMessage[i] - keyTry - 97) < 0)
                    plaintext += 123 + (encMessage[i] - keyTry - 97);
                else
                    plaintext += (encMessage[i] - keyTry - 97)%26 + 97;
            }
            else
                plaintext += encMessage[i];
        }
        cout << "BRUTE-FORCE ATTACK (DECRYPTION) - The clear text for key
        -> " << keyTry << " :- " << plaintext << endl;
    }
}

int main()
{
     int encKey;
     string cleartext;
     cout << "ENCRYPTION - Enter the text for encryption -> ";
     getline(cin,cleartext);

     cout << "Enter the key for encryption the text -> ";
     cin >> encKey;
```

```
    string encryptedMessage = encrypt(cleartext,encKey);
    cout << "ENCRYPTED MESSAGE - The encrypted message is -> "
    << encryptedMessage << endl << endl;

    //** brute force attack
    decrypt(encryptedMessage);
}
```

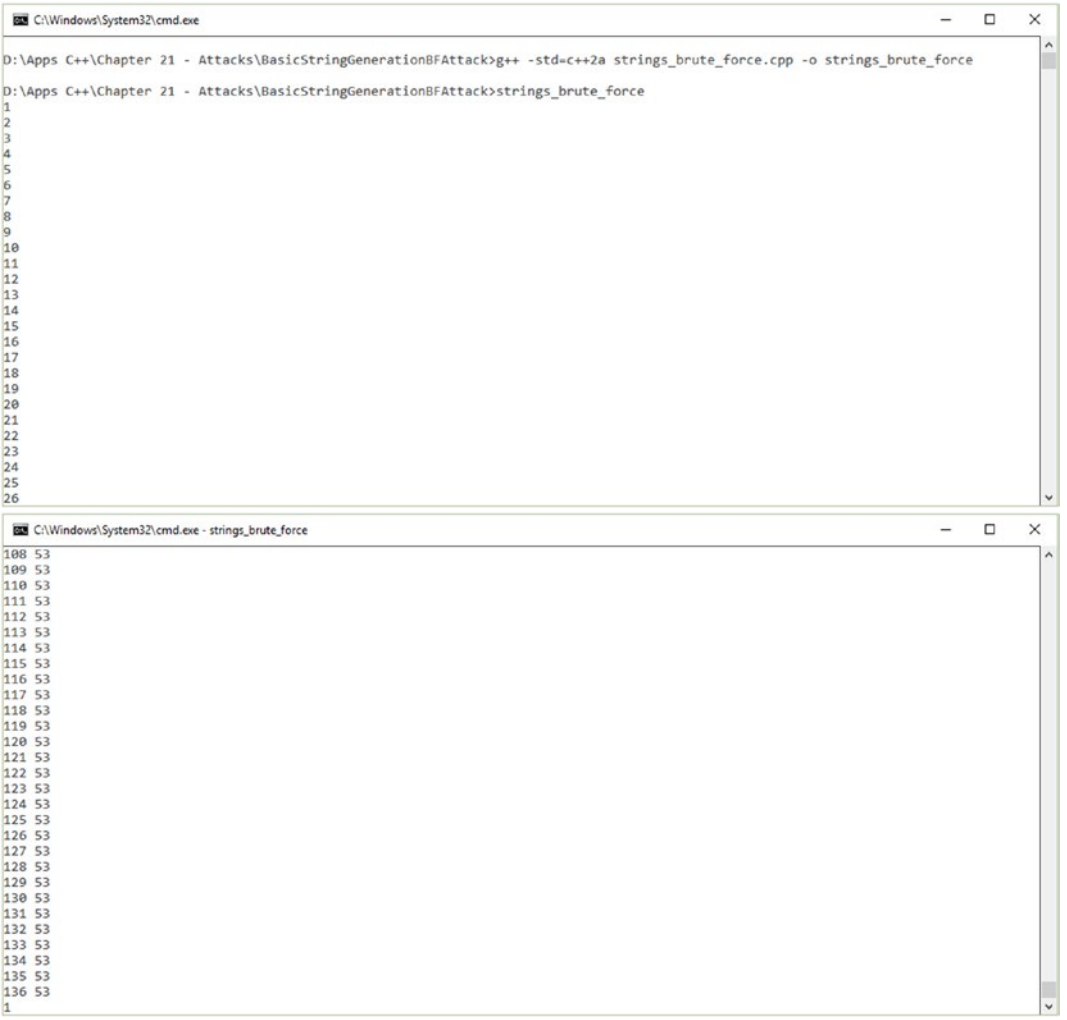


Figure 21-2. *Basic string generation for a brute-force attack (different states of generating strings)*

Listing 21-2. Basic String Generation Source Code

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

// We are using a linked list data structure.
// The reasong is to avoid some of the restrictions
//      based on the generation of the string length.
// Our list has to be converted to string in
//      such way that it can be used. The current conversion
//      might be a little slower comparing with other methods
//      due to the fact that the conversion is happening with
//      each cycle.

// Another solution consists in implementing a solution based
//      on the generation of the allocation for the string with
//      a fixed size equal with 20 characters (which is more than
//      enough.

// the structure definition for holding the characters (strings)
typedef struct charactersList charlist_t;
struct charactersList
{
     // the character
    unsigned char character;
     // the next character
    charlist_t* nextCharacter;
};

// The method will return a new initialized charlist_t element.
// The element returned is charlist_t
charlist_t* new_characterList_element()
{
    charlist_t* elementFromTheList;
```

```
     if ((elementFromTheList = (charlist_t*) malloc(sizeof(charlist_t))) != 0)
    {
        elementFromTheList->character = 0;
        elementFromTheList->nextCharacter = NULL;
    }
    else
    {
        perror("The allocation using malloc() has failed.");
    }

    return elementFromTheList;
}

 // allocation free memory by the characters list
 // listOfCharacters - represents a pointer for the first element within
the list
void freeAllocation_CharactersList(charlist_t* listOfCharacters)
{
    charlist_t* currentCharacter = listOfCharacters;
    charlist_t* nextCharacter;

    while (currentCharacter != NULL)
    {
        nextCharacter = currentCharacter->nextCharacter;
        free(currentCharacter);
        currentCharacter = nextCharacter;
    }
}

// the function display the current list of characters
// the function will iterate through the whole list and it will print all
the characters
void showCharactersList(charlist_t* list)
{
    charlist_t* nextCharacter = list;
```

```
    while (nextCharacter != NULL)
    {
        printf("%d ", nextCharacter->character);
        nextCharacter = nextCharacter->nextCharacter;
    }
    printf("\n");
}

// the function will return the next sequence of characters.
// the characters are treated as numbers 0-255
// the function proceeds by incrementation of the character from the first
   position
void nextCharactersSequence(charlist_t* listOfCharacters)
{
    listOfCharacters->character++;
    if (listOfCharacters->character == 0)
    {
        if (listOfCharacters->nextCharacter == NULL)
        {
            listOfCharacters->nextCharacter = new_characterList_element();
        }
        else
        {
            nextCharactersSequence(listOfCharacters->nextCharacter);
        }
    }
}

int main()
{
    charlist_t* sequenceOfCharacters;
    sequenceOfCharacters = new_characterList_element();

    // this while will work for all possibles combinations
    // this has to be stopped manually
```

```
    while (1)
    {
        nextCharactersSequence(sequenceOfCharacters);
        showCharactersList(sequenceOfCharacters);
    }

    freeAllocation_CharactersList(sequenceOfCharacters);
}
```

Buffer Overflow Attack

A *buffer* represents a temporary area that is used for storing data. At the moment when more data is placed by the programs or system processes, an extra data overflow will occur.

In a *buffer-overflow attack,* the extra data being stored can store specific instructions to take actions designed by hackers or malicious users. As an example, the data could trigger an event (function or process) to destroy files or reveal private data about users.

The attacker uses a buffer overflow to get an advantage from a program that is being executed and is waiting for user interaction. There are two types of buffer overflows: *stack-based* and *heap-based*. In a *heap-based* overflow, it is very difficult to launch and execute attacks based on flooding the memory space reserved for the program and its execution. In a *stack-based* overflow, the exploitation of the applications and programs is done on the memory stack, which is the memory space used to store the input data from the user.

The following example (see Figure 21-3 and Listing 21-3) shows the danger of such situations for C++ applications. In the example provided, we won't do any implementation for malicious code injection. We show the main process for buffer overflow. As a comparison between modern compilers vs. old compilers, modern compilers provide options for overflow checking during the compile or linking process but at the running time it is quite difficult to check the situation without having a protection mechanism such as the handling process of the exceptions.

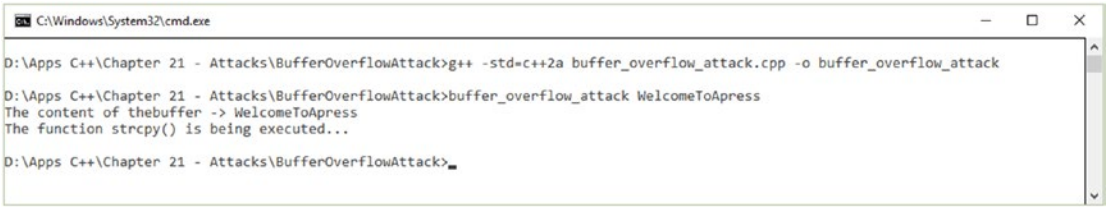


Figure 21-3. *Buffer overflow execution*

Listing 21-3. Implementation of the Buffer Overflow Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        // We allocate a buffer of 5 bytes which includes also the
           termination, NULL.
        // The allocation should be done as 8 bytes which is two double
           words.
        // For overflowing process we will need more than 8 bytes.

        // if the user provides more than 8 characters for the input,
        // an access violation and fault segmentation
       char buffer_test_example[5];
        // execution of the program
       if (argc < 2)
       {
              printf("Function strcpy() will not be executed...\n");
              printf("The syntax: %s <characters>\n", argv[0]);
              exit(0);
       }

        // Take the input from the user and copy it to the buffer.
        // The process is done without verifying the bound
       strcpy(buffer_test_example, argv[1]);
       printf("The content of thebuffer -> %s\n", buffer_test_example);
```

```
        printf("The function strcpy() is being executed...\n");

        return 0;
}
```

Conclusion

The current chapter covered two important attacks, brute-force attacks and buffer overflow attacks. You are now capable of

- Understanding the brute-force and buffer overflow attacks
- Understanding the main concepts that form the basics of designing such attacks
- Understanding the limitations between stack-based and heap-based buffer overflows

References

[1] M. I. Mihailescu and S. Loredana Nita, "Three-Factor Authentication Scheme Based on Searchable Encryption and Biometric Fingerprint" in *2020 13th International Conference on Communications (COMM)*. Bucharest, Romania, 2020, pp. 139-144, doi: 10.1109/COMM48946.2020.9141956.

[2] M. I. Mihailescu, S. L. Nita, and V. C. Pau. "Applied Cryptography in Designing e-Learning Platforms" in *16th International Scientific Conference "eLearning and Software for Education."* Bucharest, Apr. 2020, vol. 2, pp. 179–189, doi: 10.12753/2066-026X-20-108.

CHAPTER 22

Text Characterization

In this chapter, we will analyze two important metrics for cipher and plaintext analysis: the chi-squared statistic and searching for patterns (monograms, bigrams, and trigrams). When working with classic and modern cryptography, text characterization as technique is a very important part of the cryptanalysis backpack.

The Chi-Squared Statistic

The chi-squared statistic is an important metric that computes the similarity percent between two probability distributions. There are two situations when the result of the chi-squared statistic is equal to 0; it means that the two distributions are similar. If the distributions are very different, a higher number will be outputted.

The chi-squared statistic is defined by the following formula:

$$\chi^2(C,E)=\sum_{i=A}^{i=Z}\frac{(C_i-E_i)^2}{E_i}$$

In the following example (see Listing 22-1), we will compute an example of a chi-squared distribution.

Listing 22-1. Chi-Squared Distribution Source Code

```
#include <iostream>
#include <random>

int main()
{
  const int number_of_experiments=10000;
  const int number_of_stars_distribution=100;    // maximum number of stars
                                                 to distribute
```


M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_22

```
  std::default_random_engine theGenerator;
  std::chi_squared_distribution<double> theDistribution(6.0);

  int p[10]={};

  for (int i=0; i<number_of_experiments; ++i)
  {
    double no = theDistribution(theGenerator);
    if ((no>=0.0)&&(no<10.0)) ++p[int(no)];
  }

  std::cout << "chi_squared_distribution (6.0):" << std::endl;

  for (int i=0; i<10; ++i) {
    std::cout << i << "-" << (i+1) << ": ";
    std::cout << std::string(p[i]*number_of_stars_distribution/number_of_
    experiments,'*') << std::endl;
  }

  return 0;
}
```

The output is shown in Figure 22-1.

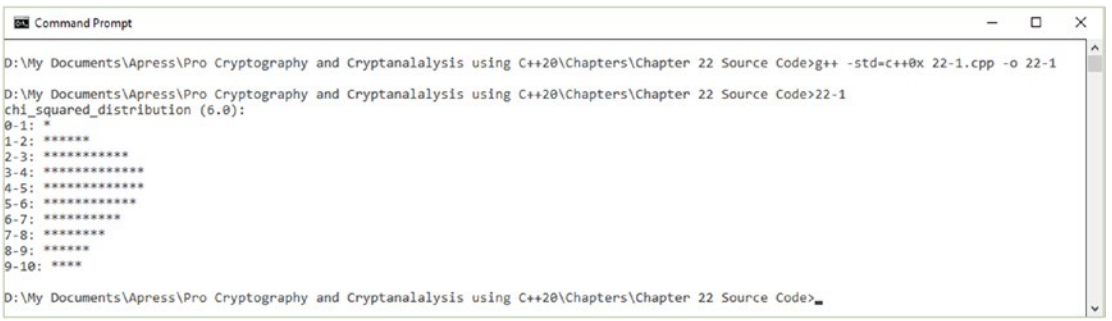


Figure 22-1. *Output of the chi-squared distribution example*

How does the chi-squared distribution example help in cryptanalysis and cryptography?

The first step that must be made is to compute the frequency of the characters within the ciphertext. The second step is to compare the frequency distribution of the assumed language that is used for encryption (e.g. English) with shifting the two

frequency distributions related to one another. In this way we have the chance to find the *shift* that was used during the encryption process. This procedure is a standard and simple procedure that can be used on ciphers, such as the Caesar cipher. This take place when the frequency of English characters is lined up with the frequency of a ciphertext. Figure 22-2 shows the frequencies of the occurrences for English characters.

As an example, let's consider the following example encrypted with a Caesar cipher, which has 46 characters:

```
ZHOFRPHWRDSUHVVWKLVLVHQFUBSWHGZLWKFDHVDUFLSKHU
```

It is very important to understand that the chi-squared statistic is based on counts and not on probabilities. For example, if we have the letter E, with its occurrence probability of 0.127, the expectation is that the occurance will be 12.7 times within 100 characters.

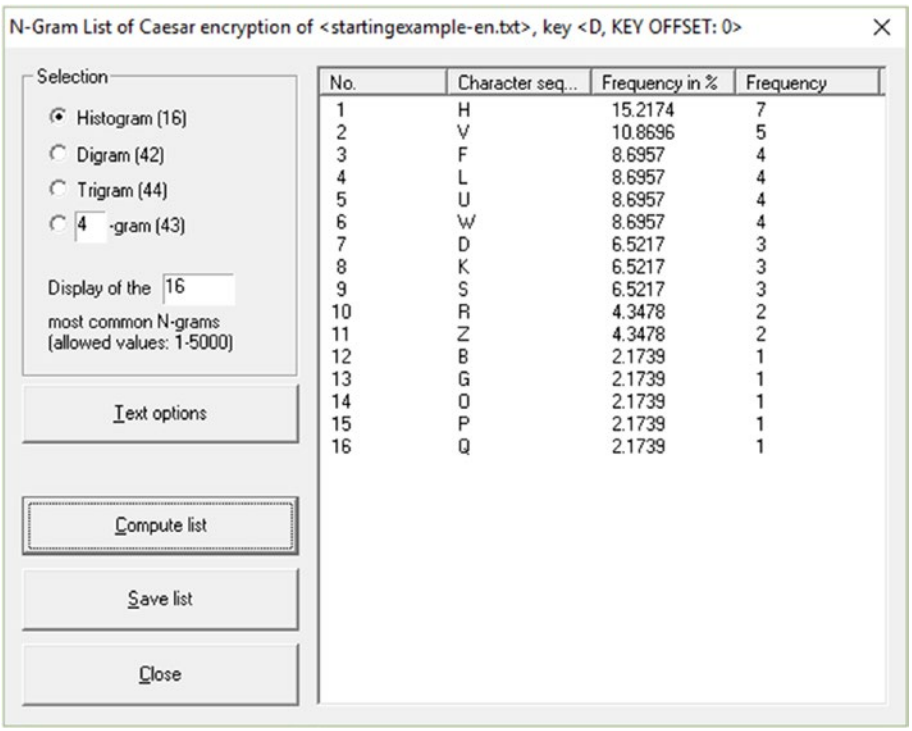


Figure 22-2. *Letter frequency for encrypted text*

To compute the count expected, the length of the ciphertext must be multiplied with the probability. The cipher from above has a total of 46 characters. Following the statistic with E from above, the expectation is that the E letter occurs $46 \cdot 0.127 = 5.842$ times.

In order to solve the Caesar cipher, we need to use each of the possible 25 possible keys, using the letter or the position of the letter within the alphabet. For this, it's very important how the count starts: from 0 or from 1. the chi-squared must be computed for each of the keys. The process consists of comparing the count number of the letter with what we can expect the counts to be if the text is in English.

To compute the chi-squared statistic for our ciphertext, we will count each letter. We find that the letter H occurs seven times. If the language used is English, it should appear $46 \cdot 0.082 = 3.772$ times. Based on the output, we can compute the following:

$$\frac{(7-3.772)^2}{3.772} = \frac{3.228^2}{3.772} = \frac{10.420}{3.772} = 2.762$$

This procedure is done for the rest of letters and making addition between all the probabilities (see Figure 22-3).

Once the ciphertext is decrypted, the plaintext should be

```
WELCOMETOAPRESSTHISISENCRYPTEDWITHCAESARCIPHER
```

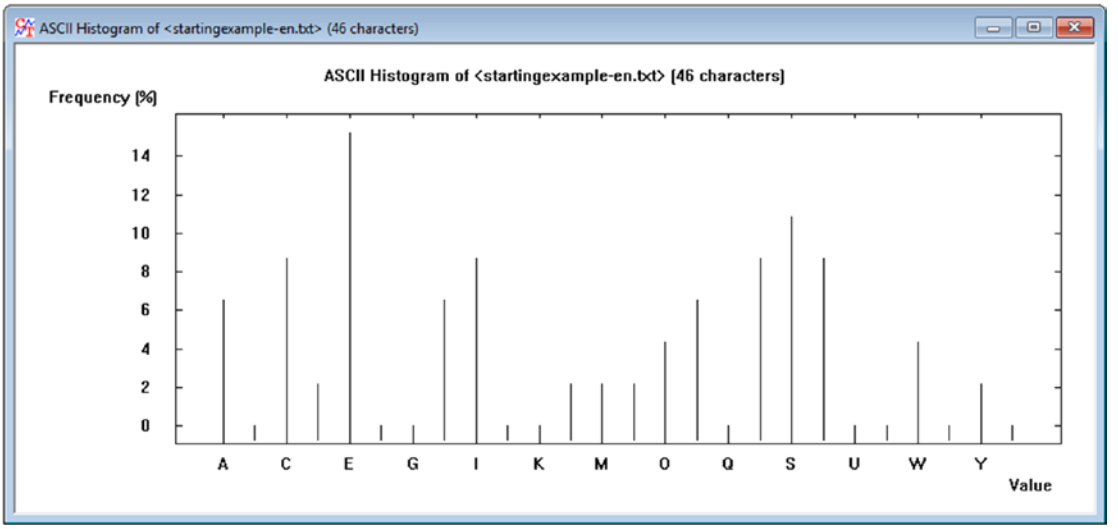


Figure 22-3. *Encryption letter frequency (%)*[1]

[1]The letter encryption frequency is generated using CrypTool, www.cryptool.org/en/

Cryptanalysis Using Monogram, Bigram, and Trigram Frequency Counts

Frequency analysis is one of the best practices for finding the occurrence of characters in a ciphertext, with the goal of breaking the cipher. The analysis, based on pattern analysis, can be used to measure and count the characters as bigrams (or digraphs), a method for measuring pairs of characters that occur within the text. Trigram frequency analysis measures the occurrence of combinations formed out of three letters.

In this section, we will focus on text characterization with bigrams and trigrams that can be used for resolving ciphers, such as Playfair.

Counting Monograms

Counting monograms is one of the most effective methods used in substitution ciphers, such as Caesar ciphers, Polybius squares, and so on. The method works very well because the English language has a specific frequency distribution. This also means that is not hidden by substitution ciphers. The distribution is shown in Figure 22-4.

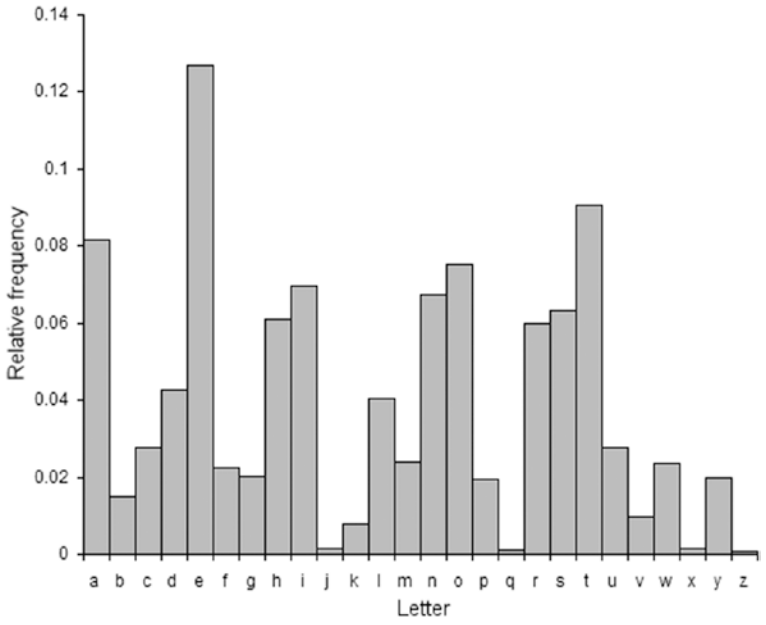


Figure 22-4. *Letter frequency for the English language*

Counting Bigrams

Bigrams counting is based on the same idea as counting monograms. Instead of counting the occurrence of single characters, you count the occurrence frequency for pairs of characters.

Figure 22-5 lists some of the common bigrams experienced during the cryptanalysis process. In Listing 22-2, we implemented a solution that counts the occurrences of bigrams. Figure 22-6 shows the output of this example of counting the bigrams. The source code from Listing 22-2 uses a file called `bigram.txt` which contains a sample text to illustrate the process of counting the bigrams.

```
TH 11699784
HE 10068926
IN 87674002
ER 77134382
AN 69775179
RE 60923600
ES 57070453
ON 56915252
ST 54018399
NT 50701084
EN 48991276
AT 48274564
ED 46647960
ND 46194306
TO 46115188
OR 45725191
EA 43329810
TI 42888666
AR 42353262
TE 42295813
NG 38567365
AL 38211584
IT 37938534
AS 37773878
IS 37349981
HA 35971841
ET 32872552
SE 31532272
OU 31112284
OF 30540904
```

Figure 22-5. *Bigrams*

Figure 22-6. *Counting bigrams*

Listing 22-2. Counting Bigrams

```
#include <stdio.h>

int main(void)
{
    int alphabet_counting['z' - 'a' + 1]['z' - 'a' + 1] = {{ 0 }};
    int character0 = EOF, character1;
    FILE *fileBigramSampleText = fopen("bigram.txt", "r");

    if (fileBigramSampleText != NULL)
      {
        while ((character1 = getc(fileBigramSampleText)) != EOF)
          {
            if (character1 >= 'a' && character1 <= 'z' && character0 >= 'a'
            && character0 <= 'z')
             {
                alphabet_counting[character0 - 'a'][character1 - 'a']++;
            }
            character0 = character1;
        }
        fclose(fileBigramSampleText);
        for (character0 = 'a'; character0 <= 'z'; character0++)
            {
            for (character1 = 'a'; character1 <= 'z'; character1++)
                {
```

```
                int number = alphabet_counting[character0 - 'a']
                [character1 - 'a'];
                if (number)
                    {
                    printf("%c%c: %d\n", character0, character1, number);
                }
            }
        }
    }
    return 0;
}
```

Listing 22-3 and Figure 22-7 show a more general version that is designed to handle 8-bit character pairs.

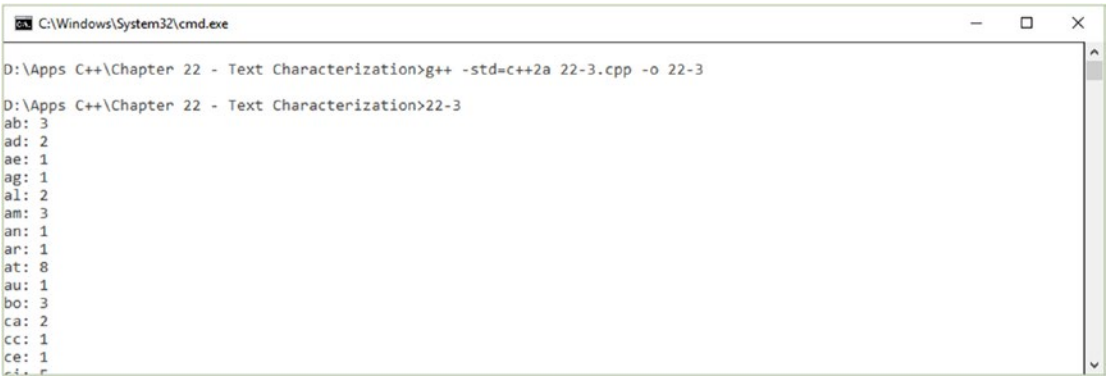


Figure 22-7. *Output for 8-bit character pairs*

Listing 22-3. General Version for Working with 8-bit Character Pairs

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    // the last five bytes corresponds to ISO/IEC 8859-9
    const char alphabet[] = "abcdefghijklmnopqrstuvwxyz\xFD\xFxE7\xF6\xFC";
    const int length_of_alphabet = (sizeof(alphabet) - 1);
    int count[length_of_alphabet][length_of_alphabet];
```

```
    char *position0 = NULL;
    int character1;
    FILE *fileTextForCountingBigrams = fopen("bigram.txt", "r");

    memset(count, 0, sizeof(count));

    if (fileTextForCountingBigrams != NULL)
       {
        while ((character1 = getc(fileTextForCountingBigrams)) != EOF)
           {
            char *p1 = (char*)memchr(alphabet, character1, length_of_alphabet);
            if (p1 != NULL && position0 != NULL)
               {
                count[position0 - alphabet][p1 - alphabet]++;
            }
            position0 = p1;
        }
        fclose(fileTextForCountingBigrams);
        for (size_t i = 0; i < length_of_alphabet; i++)
            {
            for (size_t j = 0; j < length_of_alphabet; j++)
                {
                int n = count[i][j];
                if (n > 0)
                   {
                    printf("%c%c: %d\n", alphabet[i], alphabet[j], n);
                }
            }
        }
    }
    return 0;
}
```

Counting Trigrams

Trigrams counting operates on the same principle as bigram counting; the difference consists in counting three characters.

Figure 22-8 lists some of the common trigrams experienced during the cryptanalysis process. In Listing 22-4, we implement a solution for finding and counting the occurrences of trigrams within a text (see Figure 22-9). The solution is different from the ones in Listing 22-2 and Listing 22-3.

```
THE 77534223
AND 30997177
ING 30679488
ENT 17902107
ION 17769261
HER 15277018
FOR 14686159
THA 14222073
NTH 14115952
INT 13656197
ERE 13287155
TIO 13285065
TER 12769843
EST 11956466
ERS 11823017
ATI 11227573
HAT 10900482
ATE 10712298
ALL 10501105
ETH 10304110
HES 10189449
VER 10156140
HIS 10051039
OFT 9434246
ITH 9142241
FTH 9036651
STH 9024058
OTH 8869058
RES 8835871
ONT 8757161
DTH 8745845
ARE 8741156
REA 8700830
EAR 8697937
WAS 8640940
```

Figure 22-8. *Trigrams*

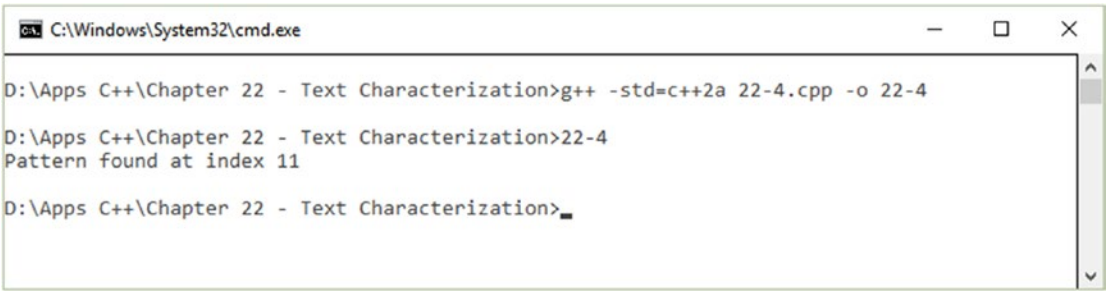


Figure 22-9. *Displaying a trigram*

Listing 22-4. Counting Trigrams

```
#include <iostream>
using namespace std;

void printTrigramOccurance(string fullText, string trigramPattern)
{
    int occurance = fullText.find(trigramPattern);
    while (occurance!= string::npos)
    {
        cout << "Pattern found at index " << occurance << endl;
        occurance = fullText.find(trigramPattern, occurance + 1);
    }
}
int main()
{
    string fullText = "Welcome to Apress.";
    string trigramPattern = "Apr";
    printTrigramOccurance(fullText, trigramPattern);
}
```

Conclusion

The chapter covered the concept of text characterization and showed its importance in the cryptanalysis process. You can now deal with the chi-squared statistic, and you can work with monograms, bigrams, and trigrams to decrypt substitution ciphertexts. As a summary, you learned about

- The concept of text characterization
- Working with monograms, bigrams, and trigrams
- Implementing the chi-squared statistic
- Monogram, bigram, and trigram implementations

References

[1] Simon Singh, *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. ISBN 0-385-49532-3. 2000

[2] Helen F. Gaines, *Cryptanalysis: A Study of Ciphers and Their Solution*. 1989.

CHAPTER 23

Implementation and Practical Approach of Cryptanalysis Methods

The current chapter is a general discussion of the methodologies of cryptanalysis methods and how those methods can be applied in a quick and efficient way. The proposed methodologies are classic and actual (modern) cryptography/cryptanalysis algorithms and methods. Quantum cryptography is not included at this moment.

The methodology proposed (see Figure 23-1) is designed with the goal of helping you, the cryptanalyst, to know where you are situated during the cryptanalysis process. You can use the map presented in Figure 23-1 to choose the proper tool or method for your work.

Proceeding with the implementation of the cryptanalysis methods can be a very laborious task if you don't have the proper information about the cryptographic method. The following will present a short process for identifying the necessary elements for conducting the cryptanalysis process. The cryptanalysis process consists of two general steps:

- *Step 1* is based on identifying what type of cryptanalysis should be conducted.
- S*tep 2* consists of gathering everything that you know about cryptography algorithms. After these two steps have been performed properly and with maximum seriousness, you can move forward with the extra steps.
- *Step 3* is when you build a proper *attack model.*
- *Step 4* is when you choose the proper tools.

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4_23

Step 1. This step deals with what kind of cryptanalysis should be performed. Within this step the cryptanalysis will decide within the business environment what role they will play: legal and authorized cryptanalyst, ethical hacker, or malicious cracker. As soon as you decide your role, you can move to Step 2.

Step 2. If the cryptanalyst is legitimate, they must be aware of two things before getting started: the *cryptography algorithm* and the *cryptographic key*. Based on the experience of some of cryptanalysts, this is not a necessary requirement but in some cases it will be very useful to know. As soon as you are aware of the cryptography algorithm and cryptographic key, you can easily start the cryptanalysis process by applying the proper methods and testing the security of the business applications.

Step 3. This step is based on setting up the attack model or attack type. Attack models and attack types will point out a quantitative variable used to indicate how much information a cryptanalyst will have access to when they perform the cracking methods on the encrypted message. The most important attacks are

- Ciphertext-only attack
- Known-plaintext attack
- Chosen-plaintext attack
- Chosen-ciphertext attack
    - Adaptive chosen-ciphertext attack
    - Indifferent chosen-ciphertext attack

Step 4. After the attack model has been picked or another model has been created and adapted properly with the case and requirements, you move to the next step, which is to pick the software tools. Choosing software tools from ones that already exist or creating your own tools can be time consuming but in practice will have massive contributions. Summarizing this, here are some tools that can be used in the cryptanalysis process, depending on what you're trying to test.

- **Penetration tools**: Kali Linux, Parrot Security, BackBox
- **Forensics**: DEFT, CAINE, BlackArch, Matriux
- **Databases**: sqlmap (standalone version), Metasploit framework (standalone version), VulDB

- **Web and network**: Wireshark, Nmap, Nessus, Burp Suite, Nikto, and OpenVas
- **Other tools**: CrypTool (very useful and amazing tool)

The tools mentioned above represent a selection that are very used in practice and can produce desired results.

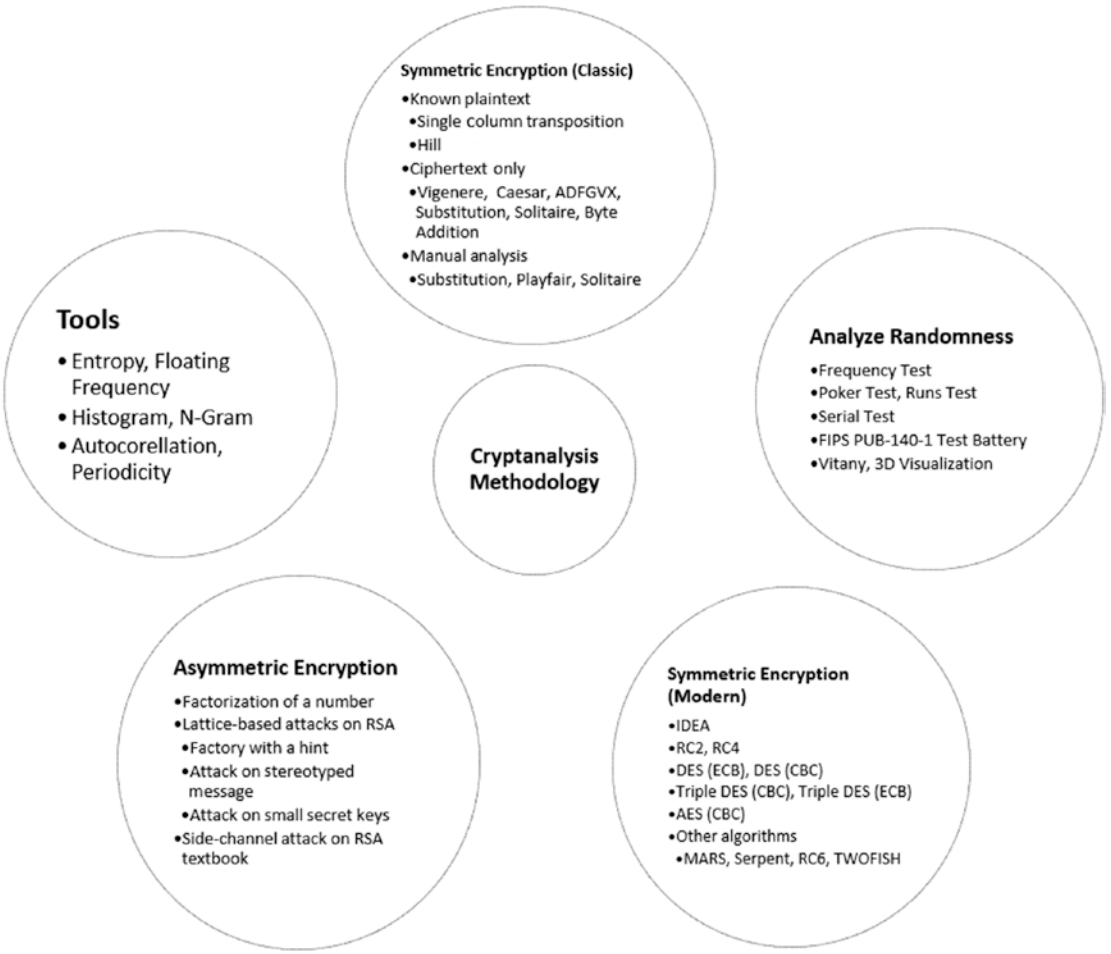


Figure 23-1. *The cryptanalysis methodology*

Ciphertext-Only Attack

A ciphertext-only attack (COA) represents the weakest attack because it can easily be used by the cryptanalyst due to the fact that they just encoded the message.

The attacker–cryptanalyst has access to a set of ciphertexts. The attack is fully successful if the corresponding plaintexts are deduced together with the key.

In this type of attack (see Figure 23-2), the attacker/cryptanalyst will be able to observe the ciphertext. Everything that the cryptanalyst will see is represented by a set of scrambled and nonsense characters that create the output based on the encryption process.

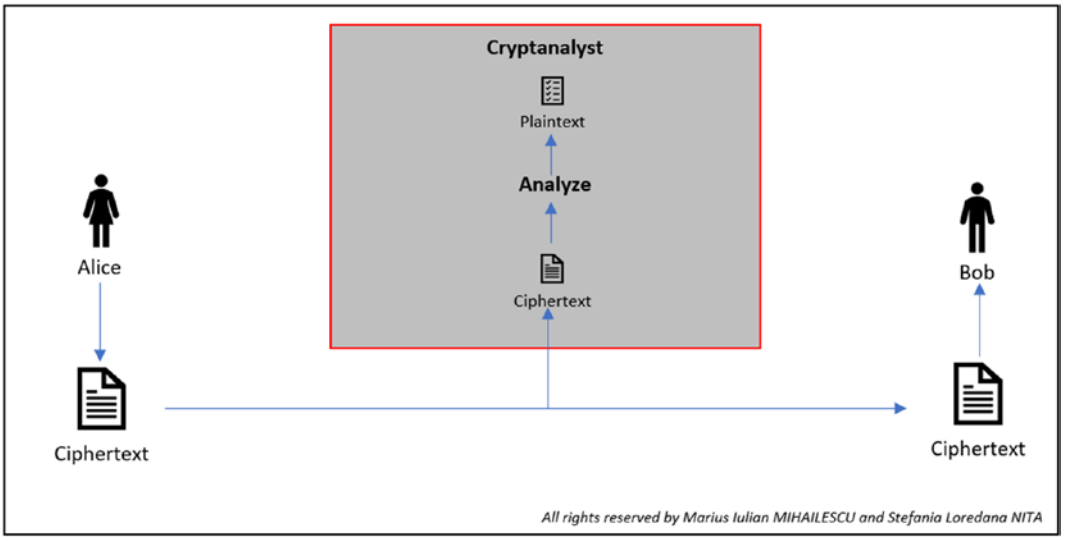


Figure 23-2. *COA representation*

Known-Plaintext Attack

The known-plaintext attack (KPA) helps the cryptanalyst to generate the ciphertext based on the fact that he is aware of the ciphertext.

The cryptanalyst follows a simple procedure by selecting the plaintext, but they will observe the pair that is compounded from the plaintext and ciphertext. The chance of success is better compared to COA. Simple ciphers are quite vulnerable to this attack. See Figure 23-3.

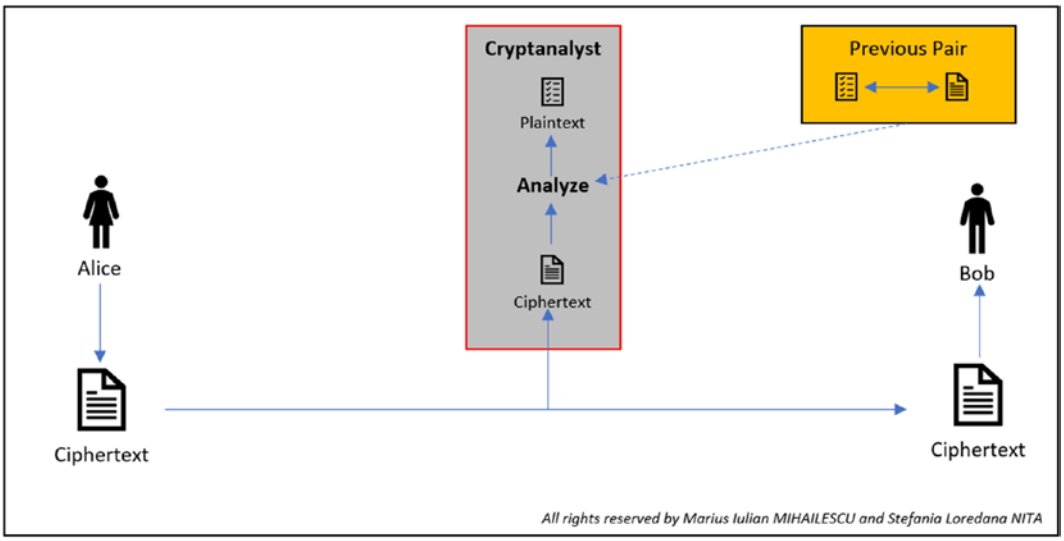


Figure 23-3. *KPA representation*

Chosen-Plaintext Attack

In a chosen-plaintext attack (CPA), the cryptanalyst has the ability to select the plaintext that has been sent encrypted using an encryption algorithm and they can observe how the ciphertext is generated. This can be observed as an active model where the cryptanalyst has the chance to select the plaintext and to realize the encryption.

Based on the ability to select and pick any plaintext, the cryptanalyst has the chance to observe vital details about the ciphertext, which gives them a strong advantage in understanding how the algorithm works inside and the chance to get the secret key.

A professional cryptanalyst will have a strong database that contains known plaintexts, ciphertexts, and possible keys. In Listing 23-1 and Figure 23-5, we have provided an example of generating possible keys automatically. It is a very simple example for illustrating how possible keys can be generated. They can be used with the pairs to determine the cipher text input (see Figure 23-4).

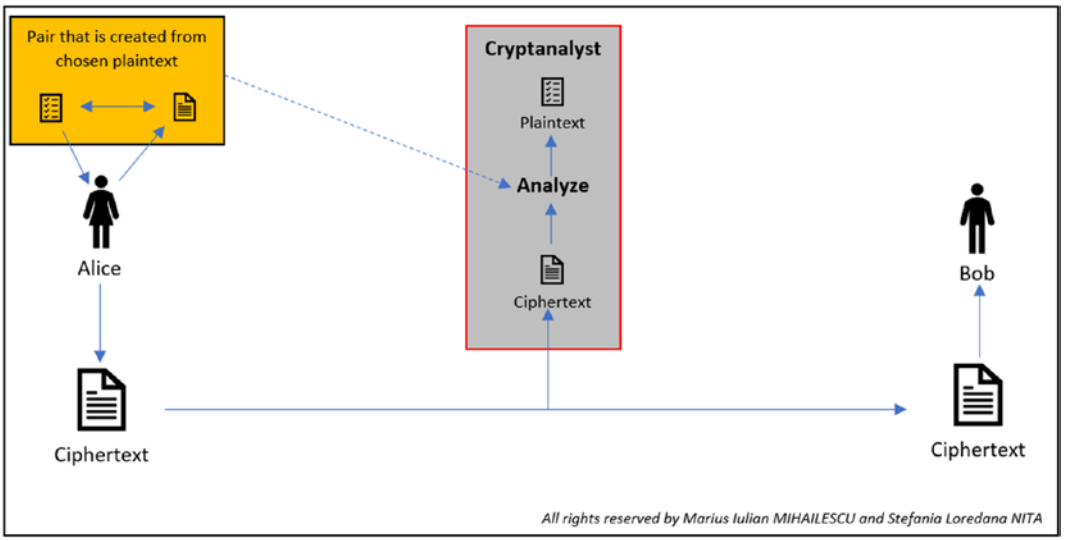


Figure 23-4. *CPA representation*

Listing 23-1. Automatic Generation of Random Keys

```
#include <stdio.h>
#include <time.h>
#include <iostream>

using namespace std;

//** generate an integer that is situated between 1 to 4
int generateInteger()     {
     //** pseudo-random generator (srand).
     //** time(NULL) represents the seed
    srand(time(NULL));

     //** generate a random value and store
     //** the remainder of rand() to 5
    int randomValue = rand() % 5;

     //** if the value is equal with 0, move to the
    //** next value of i and return that value
    if (randomValue == 0)
        randomValue++;
```

```
    return randomValue;
}

//** the function will generate randomly
//** an integer situated between 0 and 25
int generateRandomlyInteger(){
     //** pseudo-random generator (srand).
    //** time(NULL) represents the seed
    srand(time(NULL));

     //** generate a random value and store
    //** the remainder of rand() with 26
    int random_key = rand() % 26;
    return random_key;
}

//** based on the length provided, the function
//** will generate a cryptographic key
void generate_crypto_key(int length){
     //** create a string variable for cryptography
     //** key and initialize it with NULL
    string crypto_key = "";

     //** variable used for cryptography key generation
    string alphabet_lower_case = "abcdefghijklmnopqrstuvwxyz";
    string alphabet_upper_case = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    string special_symbols = "!@#$%&";
    string digits_and_numbers = "0123456789";

    //** local variables and their initializations
    int key_seed;
     int lowerCase_Alphabet_Count = 0;
     int upperCase_Alphabet_Count = 0;
     int digits_And_numbers_count = 0;
     int special_symbols_count = 0;

    //** the variable count will save the length
    //** of the cryptography key.
    //** initially we will set it to zero
```

```
    int countingLengthCryptoKey = 0;
    while (countingLengthCryptoKey < length) {
          //** generateInteger() function will return a number that
          //** is situated between 1 and 4.
          //** The number that is generated will be used in
          //** assignation with one of the strings that has been
          //** defined above (for example: alphabet_lower_case,
          //** alphabet_upper_case, special_symbols, and
          //** digits_and_numbers).
          //** This being said, the following correspondence will
          //** be applied: (1) for alphabet_lower_case, (2) for
          //** alphabet_upper_case, (3) for special_symbols, and
          //** (4) digits_and_numbers
          int string_type = generateInteger();

          //** For the first character of the cryptography key we
          //** will put a rule in such way that it should be a
          //** letter, in such way that the string that will be
          //** selected will be an lower case alphabet or an upper
          //** case alphabet. The IF condition is quite vital as
          //** the switch is based on it and the value that
          //** string_type variable will have.
          if (countingLengthCryptoKey == 0) {
              string_type = string_type % 3;
              if (string_type == 0)
                 string_type++; }

        switch (string_type) {
               case 1:
                    //** based on the IF condition, it is
                    //** necessary to check the minimum
                    //** requirements of the lower case alphabet
                    //** characters if they have been accomplished
                    //** and fulfilled. If we are dealing with the
                    //** situation in which the requirement has
                    //** not been achieved we will situate ourself
                    //** in the break phase.
```

```
        if ((lowerCase_Alphabet_Count == 2)
                  && (digits_And_numbers_count == 0
                  || upperCase_Alphabet_Count == 0
                  || upperCase_Alphabet_Count == 1
                  || special_symbols_count == 0))
             break;

        key_seed = generateRandomlyInteger();
        crypto_key = crypto_key +
                  alphabet_lower_case[key_seed];
        lowerCase_Alphabet_Count++;
        countingLengthCryptoKey++;
        break;

    case 2:
        //** based on the IF condition, it is
        //** necessary to check the minimum
        //** requirements of the upper case alphabet
        //** characters if they have been accomplished
        //** and fulfilled. If we are dealing with the
        //** situation in which the requirement has
        //** not been achieved we will situate ourself
        //** in the break phase.
        if ((upperCase_Alphabet_Count == 2)
                  && (digits_And_numbers_count == 0
                  || lowerCase_Alphabet_Count == 0
                  || lowerCase_Alphabet_Count == 1
                  || special_symbols_count == 0))
             break;
        key_seed = generateRandomlyInteger();
        crypto_key = crypto_key +
                  alphabet_upper_case[key_seed];
        upperCase_Alphabet_Count++;
        countingLengthCryptoKey++;
        break;
```

```
case 3:
     //** based on the IF condition, it is
     //** necessary to check the minimum
     //** requirements of the numbers if they have
     //** been accomplished and fulfilled. If we
     //** are dealing with the situation in which
     //** the requirement has not been achieved we
     //** will situate ourself in the break phase.
     if ((digits_And_numbers_count == 1)
               && (lowerCase_Alphabet_Count == 0
               || lowerCase_Alphabet_Count == 1
               || upperCase_Alphabet_Count == 1
               || upperCase_Alphabet_Count == 0
               || special_symbols_count == 0))
          break;
     key_seed = generateRandomlyInteger();
     key_seed = key_seed % 10;
     crypto_key = crypto_key +
               digits_and_numbers[key_seed];
     digits_And_numbers_count++;
     countingLengthCryptoKey++;
     break;

case 4:
     //** based on the IF condition, it is
     //** necessary to check the minimum
     //** requirements of the special characters if
     //** they have been accomplished and
     //** fulfilled. If we are dealing with the
     //** situation in which the requirement has
     //** not been achieved we will situate ourself
     //** in the break phase.
```

```
                if ((special_symbols_count == 1)
                        && (lowerCase_Alphabet_Count == 0
                        || lowerCase_Alphabet_Count == 1
                        || upperCase_Alphabet_Count == 0
                        || upperCase_Alphabet_Count == 1
                        || digits_And_numbers_count == 0))
                    break;

                key_seed = generateRandomlyInteger();
                key_seed = key_seed % 6;
                crypto_key = crypto_key +
                              special_symbols[key_seed];
                special_symbols_count++;
                countingLengthCryptoKey++;
                break;
        }
    }
    cout << "\n-----------------------------\n";
    cout << "      Cryptography Key         \n";
    cout << "-----------------------------\n\n";
    cout << " " << crypto_key;
    cout << "\n\nPress any key to continue... \n";
    getchar();
}

int main() {
    int option;
     int desired_length;

     //** designing the menu
    do {
        cout << "\n-------------------------------------\n";
        cout << "  Random Cryptography Key Generator    \n";
        cout << "-------------------------------------\n\n";
        cout << "    1 --> Generate a Cryptography Key"
             << "\n";
```

```
cout << "    2 --> Quit the program"
     << "\n\n";
cout << "Enter 1 for Generating Cryptograpy Key or 2
                            to quit the program  : ";
cin >> option;

switch (option) {
case 1:
    cout << "Set the length to :  ";
    cin >> desired_length;
    //** if the length entered is less than 7, an
    //** error will be shown
    if (desired_length < 7) {
        cout << "\nError Mode : The Cryptography Key
                      Length hould be at least 7\n";
        cout << "Press a key and try again \n";
        getchar();  }
    //** The desired length should bot be bigger than
    //** 100, otherwise an error will be shown
    else if (desired_length > 100)      {
        cout << "\nError Mode : The maximum length of
                  the cryptography key should be 100\n";
        cout << "Press a key and try again \n";
        getchar(); }
    //** in ohter cases, call generate_crypto_key()
    //** function to generate a cryptography key
    else
        generate_crypto_key(desired_length);
    break;
default:
    //** in case if an invalid option is entered, show
    //** to the user an error message
    if (option != 2) {
        printf("\nOups! You have entered a choice that
                                      doesn't exist\n");
```

```
                printf("Enter ( 1 ) to generate cryptography
                         key and ( 2 ) to quit the program.\n");
                cout << "Enter a key and try again \n";
                getchar();}
            break; }
    } while (option != 2);
    return 0;
}
```

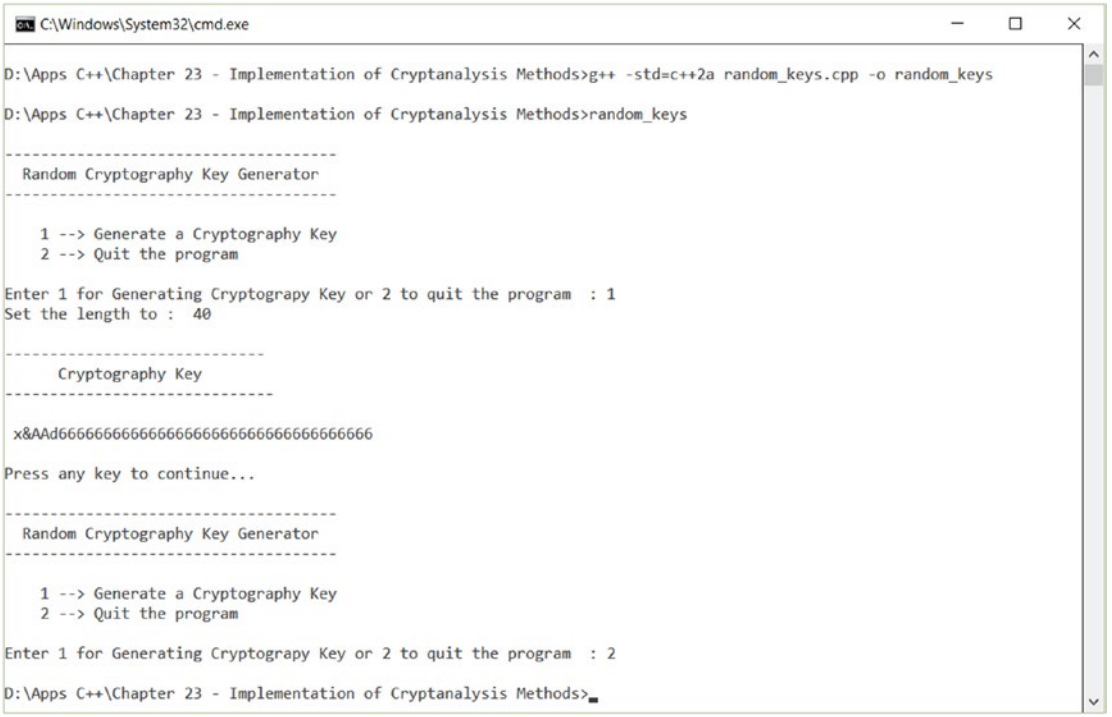


Figure 23-5. *The keys and possible passwords generated. We choose three characters to keep the processing time short*

Chosen-Ciphertext Attack

In a chosen-ciphertext attack (CCA), the cryptanalyst can perform encryption and decryption on the information. Within this attack (see Figure 23-6) the cryptanalyst can pick the plaintext, encrypt it, observe how the ciphertext is generated, and reverse the whole process.

In this attack, the cryptanalyst's mission is to find the plaintext and also to identify the algorithm and the secret key that was used for the encryption process.

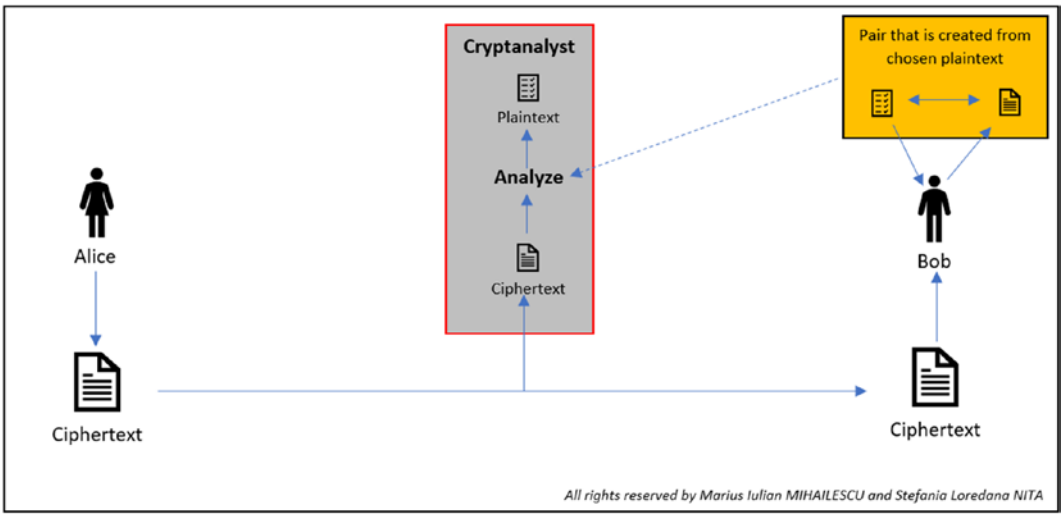


Figure 23-6. *CCA representation*

Conclusion

In this chapter, we discussed how to implement cryptanalysis methods and what defines this process for a cryptanalyst. At the end of this chapter, you now have

- A good understanding of the attack models
- The ability to follow a simple and straightforward methodology to find out where you are situated within the cryptanalysis process
- The ability to simulate and generate a database with keys and possible passwords

References

[1] Abu Yusuf Yaqub ibn Ishaq al-Sabbah Al-Kindi. Available online: www.trincoll.edu/depts/phil/philo/phils/muslim/kindi.html.

[2] Philosophers: Yaqub Ibn Ishaq al-Kindi Kennedy-Day, K. al-Kindi, Abu Yusuf Ya'qub ibn Ishaq (d. c.866–73). Available online: www.muslimphilosophy.com/ip/kin.html.

[3] Ahmad Fouad Al-Ehwany, "Al-Kindi" in *A History of Muslim Philosophy Volume 1* (pp. 421-434). New Delhi: Low Price Publications. 1961.

[4] Ismail R. Al-Faruqi and Lois Lamya al-Faruqi, *Cultural Atlas of Islam* (pp. 305-306). New York: Macmillan Publishing Company. 1986.

[5] *Encyclopaedia Britannica* (pp. 352). Encyclopaedia Britannica Inc. Chicago: William Benton. 1969.

[6] J.J. O'Connor and E.F. Robertson, *Abu Yusuf Yaqub ibn Ishaq al-Sabbah Al-Kindi*. 1999.

Index

A

B

C

M. I. Mihailescu and S. L. Nita, *Pro Cryptography and Cryptanalysis with C++20*,
https://doi.org/10.1007/978-1-4842-6586-4

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z