The Sally SERIES

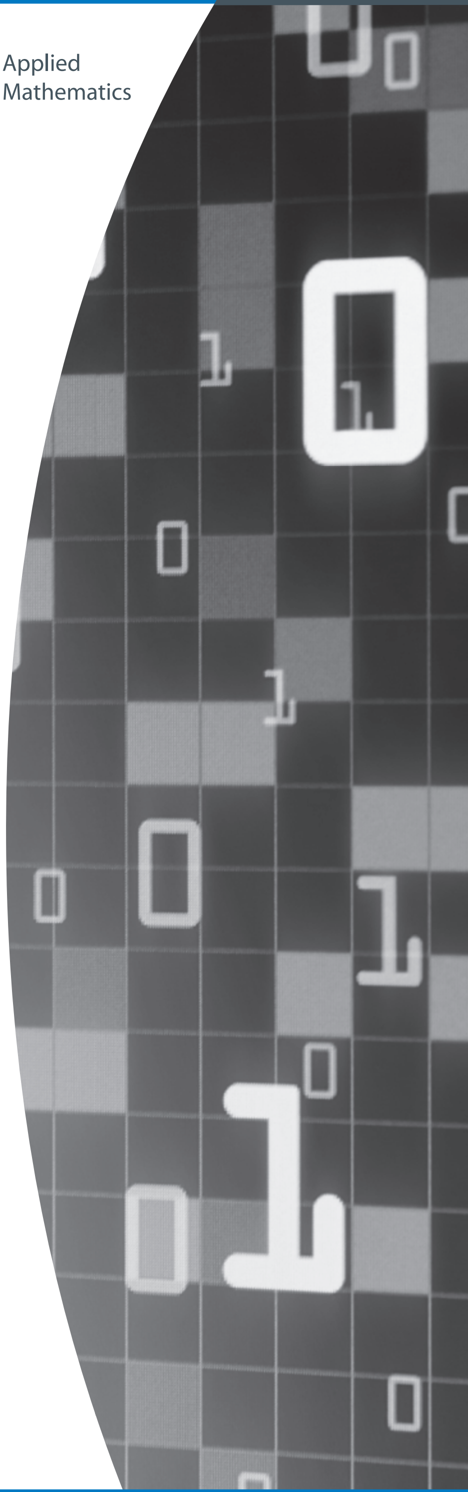
Pure and Applied
UNDERGRADUATE TEXTS

40

Applied Mathematics

A Course in Cryptography

Heiko Knospe



AMS
AMERICAN MATHEMATICAL SOCIETY

A Course in Cryptography

Heiko Knospe

The Sally Series

Pure and Applied
UNDERGRADUATE TEXTS · 40

A Course in Cryptography

Heiko Knospe

American Mathematical Society
Providence, Rhode Island USA

EDITORIAL COMMITTEE

Gerald B. Folland (Chair) Steven J. Miller
Jamie Pommersheim Serge Tabachnikov

2010 *Mathematics Subject Classification.* Primary 94A60;
Secondary 68P25, 81P94, 11T71.

For additional information and updates on this book, visit
www.ams.org/bookpages/amstext-40

Library of Congress Cataloging-in-Publication Data

Names: Knospe, Heiko, 1966– author.
Title: A course in cryptography / Heiko Knospe.
Description: Providence, Rhode Island: American Mathematical Society, [2019] | Series: Pure and applied undergraduate texts; volume 40 | Includes bibliographical references and index.
Identifiers: LCCN 2019011732 | ISBN 9781470450557 (alk. paper)
Subjects: LCSH: Cryptography–Textbooks. | Coding theory–Textbooks. | Ciphers–Textbooks. | AMS: Information and communication, circuits – Communication, information – Cryptography. msc | Computer science – Theory of data – Data encryption. msc | Quantum theory – Axiomatics, foundations, philosophy – Quantum cryptography. msc | Number theory – Finite fields and commutative rings (number-theoretic aspects) – Algebraic coding theory; cryptography. msc
Classification: LCC QA268 .K5827 2019 | DDC 005.8/24–dc23
LC record available at https://lccn.loc.gov/2019011732

Copying and reprinting. Individual readers of this publication, and nonprofit libraries acting for them, are permitted to make fair use of the material, such as to copy select pages for use in teaching or research. Permission is granted to quote brief passages from this publication in reviews, provided the customary acknowledgment of the source is given.

Republication, systematic copying, or multiple reproduction of any material in this publication is permitted only under license from the American Mathematical Society. Requests for permission to reuse portions of AMS publication content are handled by the Copyright Clearance Center. For more information, please visit `www.ams.org/publications/pubpermissions`.

Send requests for translation rights and licensed reprints to `reprint-permission@ams.org`.

© 2019 by the author. All rights reserved.
Printed in the United States of America.

∞ The paper used in this book is acid-free and falls within the guidelines established to ensure permanence and durability.
Visit the AMS home page at `https://www.ams.org/`

10 9 8 7 6 5 4 3 2 1 24 23 22 21 20 19

To my parents, Anne Anita and Karlheinz

Contents

Preface

Why is cryptography interesting? Firstly, cryptography is a classical subject with a fascinating history. Encryption techniques have been used since ancient times, but the protection against exposure was sometimes dubious and many ciphers were broken. The design of new ciphers and the capability to analyze and break them co-evolved over time. Since then, the techniques have been adapted to progress in cryptanalysis and to the computing power available. Modern cryptography goes beyond confidentiality, also addressing aspects such as data integrity, authentication, non-repudiation and other security objectives. The subject now includes all mathematical techniques relating to information security.

Secondly, cryptography is closely connected to several fields of mathematics and computer science, providing interesting applications for many theoretical results and stimulating mathematical research. Cryptography, which we use as an umbrella term for the field including cryptanalysis, is linked to aspects of discrete mathematics, number theory, algebra, probability theory, statistics, information and coding theory.

Thirdly, cryptography has become a key technology that is used ubiquitously in computer systems from small devices to large servers and networks. The multitude of security threats is driving the trend to protect data whenever possible, be it for storage or during transmission over networks. The reader should be warned, however, that real-world security is a complex process in which cryptography contributes “only” the primitives, algorithms and schemes. In practice, attacks often exploit protocol or implementation flaws, weak passwords or negligent users. Furthermore, the security guarantees offered by cryptography cannot be unconditional. The security provided depends on the power of adversaries, their computing resources and the time available for an attack, as well as on underlying computational problems which are believed to be hard.

The aim of this book is to explain the current cryptographic primitives and schemes and to outline the essential mathematics required to understand the building blocks and assess their security. We cover the widespread schemes, but we also want to address some of the recent developments in post-quantum cryptography. The mathematical and, in particular, algebraic and number-theoretical foundations of cryptography are explained in detail. The mathematical theory is presented with a focus on cryptographic applications and we do not strive for maximal generality. We look at a selection of cryptographic algorithms according to their current and supposed future relevance, while leaving out several historic schemes. Since cryptography is a very active field, some uncertainty regarding future developments will of course remain.

Why write *yet another* textbook on cryptography? We hope to convince potential readers by listing some of the unique features of this book:

- The fundamentals of cryptography are presented with rigorous definitions while being accessible to undergraduate students in science, engineering and mathematics;
- Formal definitions of security as used in the modern literature on cryptography;
- Focus on widely used methods and on prospective cryptographic schemes;
- Introduction to quantum computing and post-quantum cryptography;
- Numerical examples and SageMath (Python) code.

Cryptography can easily be underestimated by mathematicians. Several textbooks contain excellent descriptions of the mathematical theory, but fall short of explaining how to use these algorithms in practice. In fact, the main purpose of cryptography is to achieve security objectives such as confidentiality and integrity in the presence of powerful adversaries. Well-known schoolbook algorithms like RSA can be insecure without adaptations, for example, by incorporating random data.

This book follows the *provable security* approach which is adopted in the modern literature. Well-defined experiments (games) are used in which the success probability of potential attackers determines the security. Secure schemes have the property that an adversary with restricted resources can do little better than randomly guess the secret information. Using this approach, the security is reduced to standard assumptions that are generally believed to be true. In this book, we give exact security definitions and some proofs, but refer to the literature for more advanced proofs and techniques, for example, using the sequence of games approach.

We find that examples are very helpful and include computations using the open source mathematics software SageMath (aka Sage) [**Sag18**]. SageMath contains many algebraic and number theoretic functions which can be easily used and extended. Although the software might be better known among mathematicians than scientists and engineers, it is easily accessible and very suitable for cryptographic computations. SageMath is based on Python and contains other open source software as, for example, Singular, Maxima, PARI, GAP, NumPy, SciPy, SymPy and R. In recent years, Python has gained immense popularity among scientists. One of its advantages is that results can be obtained quickly without much programming overhead. In this book, we opt for SageMath instead of plain Python since SageMath has much better support for algebraic computations, which are often needed in modern cryptography. SageMath also has a convenient user interface and supports the popular Jupyter browser notebooks.

Numerical examples can be used to help understand cryptographic constructions and their underlying theory. Toy examples, in which the numbers and bit-lengths are too small for any real-world security, can still be useful in this respect. The reader is encouraged to perform computations and to write their own SageMath functions. We also provide exercises with both theoretical and numerical problems.

The book should be accessible to mathematics, science or engineering students after completing a first year's undergraduate course in mathematics (calculus and linear algebra). The material originates from several courses on cryptography for computer scientists and communication engineers which the author has taught. Since the previous knowledge can be quite heterogeneous, we decided to include several elementary topics. In the author's teaching experience, abstract algebra as well as linear algebra over general fields deserves special attention. Linear maps over finite fields play an important role in many cryptographic constructions. This book should be largely self-contained and requires no previous knowledge of discrete mathematics, algebra, number theory or cryptography. We do not strive for greatest generality and frequently refer to more specialized textbooks or articles.

Cryptography can be taught at different levels and to different audiences. This book can be used in bachelor's and master's courses, as well as by practitioners, and is suitable for a general audience wanting to understand the fundamentals of modern cryptography. Many mathematics and computer science students may already have the necessary background in discrete mathematics, elementary number theory and probability and can therefore skip Chapters 1 and 3. Chapter 4 provides the necessary algebraic constructions and is recommended to all readers without solid knowledge of abstract algebra. From my teaching experience, algebra can be a major stumbling block and should not be underestimated. Chapters 1, 3 and 4 thus provide the mathematical background of cryptography.

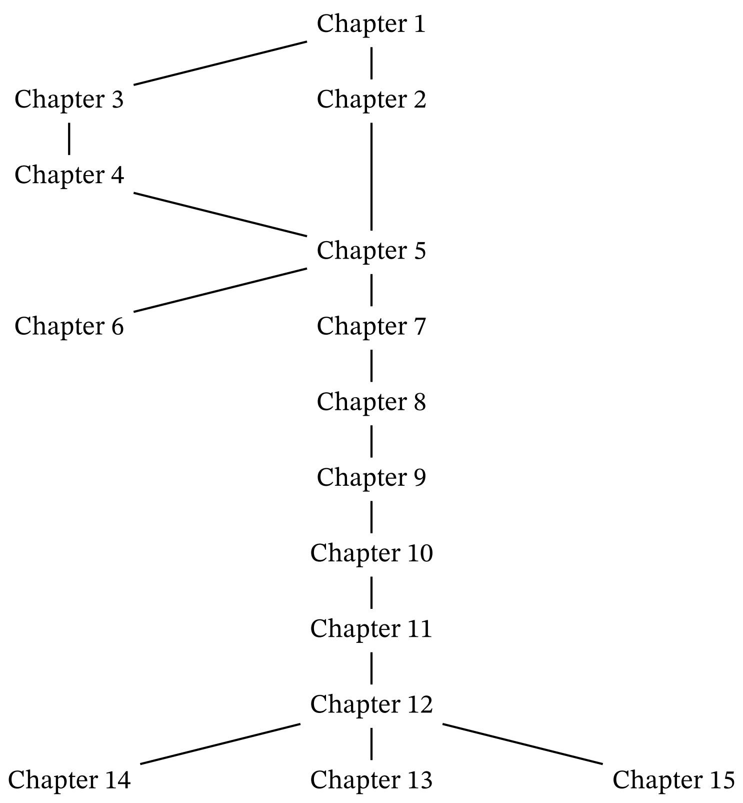
We decided to begin with the core cryptographic content as early as possible, so Chapter 2 deals with encryption schemes and the modern definitions of security. This chapter requires only basic discrete mathematics, complexity and probability theory and is recommended for most readers, even if they have some prior knowledge of cryptography. Understanding the provable security approach is crucial for the subsequent chapters of this book. Chapter 5 deals with block ciphers and AES in particular, which is a crucial part of every modern course on cryptography. Chapter 6 explores stream ciphers, which form a natural complement, but it is also possible to omit this chapter if you are short on time. We have already mentioned that modern cryptography goes beyond encryption. Integrity protection is another major objective, and hash functions and message authentication codes play a crucial role in this. These topics are addressed in Chapters 7 and 8. Chapters 9, 10 and 11, which are on public-key encryption, key establishment and signatures, introduce the fundamentals of public-key cryptography. We explain RSA and Diffie-Hellman in particular and discuss their security, which is based on hard number-theoretic problems.

We therefore think that Chapters 2, 5, 7, 8, 9, 10 and 11, along with the necessary mathematical preparations (Chapters 1, 3 and 4), should be covered in every first course on cryptography. A one-semester bachelor's module might end after Chapter 11, but whenever possible, we recommend including Chapter 12 on elliptic curve cryptography. This has been the topic of intensive research in the last few decades but has now become part of well-established cryptography and is implemented by every Internet browser, for example. We believe the basics of elliptic curves are accessible to readers after the preparatory work in Chapters 3 and 4. There are, however, more advanced topics in elliptic curves that are not treated here.

Chapters 13, 14 and 15 provide an introduction to the new field of post-quantum cryptography. In Chapter 13, we explore the basics of quantum computing and explain why quantum computers can break classic public-key schemes like RSA. Chapters 14 and 15 deal with two major types of post-quantum systems that are based on lattices and error-correcting codes, respectively. We focus on the foundations and several selected encryption schemes. Note that there are other post-quantum systems, for example, cryptosystems from isogenies of elliptic curves or multivariate-quadratic-equations signatures, which are not covered in this book. Chapters 13–15 are more challenging with respect to the level of calculus and abstract algebra. However, we spend some time on examples (many of them using SageMath) and we hope that the content of these three chapters is accessible for master's or advanced bachelor's students. We expect that quantum computing and post-quantum schemes will become increasingly important in the future.

I would be happy to receive feedback and suggestions for improvement. Please email your comments to `heiko.knospe@th-koeln.de`. Updates and additional material, for example, solutions to selected exercises and SageMath code, are available on the following website: `https://github.com/cryptobook`.

Finally, I would like to thank my colleagues and my students for their valuable feedback on my cryptography course and on earlier versions of the manuscript.



Dependence relationship between the chapters.

Getting Started with SageMath

SageMath (aka Sage) is an open source mathematics software which is ideally suited for cryptography. SageMath supports a lot of algebraic constructions used in cryptography, and results can be achieved with relatively few lines of code. Many experts in the field use SageMath for their research and for prototyping before finally switching to faster programming languages like C and C++.

This book contains a large number of examples and exercises which use SageMath, and readers are encouraged to do their own experiments. The aim of this chapter is to give a brief introduction to SageMath. Further information and links to online documentation can be found on `http://www.sagemath.org/help.html`. We also recommend the book [**Bar15**].

0.1. Installation

The installation of SageMath is easy using pre-built binaries which are available from `http://www.sagemath.org/download.html` for several types of CPU and operating systems including Linux (Ubuntu, Debian), macOS and Microsoft Windows. A Docker container is also available. Since Sage 8.0, Windows users can use a binary installer (instead of a virtual machine) and run either the SageMath shell or the browser-based interface by clicking on the respective icon. macOS users can install the `app.dmg` package, move the software to the `Applications` folder and start SageMath either from the command line or by clicking on the app icon. On Linux, the downloaded package has to be uncompressed (using `bunzip2` and `tar`). The `SageMath` directory contains an executable file that can be started from the command line. The directory can be moved if desired. It is advisable to add the directory to the local `PATH` variable or otherwise specify the full path when `sage` is executed. The `sage` executable starts a shell and `sage -notebook=jupyter` runs the Jupyter notebook server and opens a browser window.

The packages and the installed software require several gigabytes of free disk space. The SageMath distribution includes a long list of open source software and it is not usually necessary to install additional packages.

0.2. SageMath Command Line

SageMath has several user interfaces: the SageMath command line shell and browser-based notebook interfaces. The `sage` command (on macOS and Linux) and the SageMath shell icon (for Windows) starts the command line interface with access to all of its capabilities, including graphics.

```
sage$ /Applications/SageMath-8.5.app/sage
SageMath version 8.5, Release Date: 2018-12-22
Using Python 2.7.15. Type "help()" for help.
sage:
```

SageMath can be used as a calculator which supports many algebraic and symbolic computations. By default, SageMath works over the integers $\mathbb{Z}$, but the many other base rings (for example polynomial rings) and fields (in particular $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ and finite fields) are also supported.

SageMath can factorize relatively large numbers:

```
sage: factor(2^128-1)
3 * 5 * 17 * 257 * 641 * 65537 * 274177 * 6700417 * 67280421310721
```

In the following example, we define a 3×3 matrix over the integers and compute the determinant and the inverse matrix.

```
sage: A=matrix([[1,2,3],[-1,3,4],[2,2,3]])
sage: det(A)
-1
sage: 1/A
[ -1   0   1]
[-11   3   7]
[  8  -2  -5]
```

0.3. Browser Notebooks

Browser-based notebooks are a convenient user interface and are recommended for working on the examples and exercises in this book. The command

```
sage -notebook=jupyter
```

(or clicking the SageMath icon) starts a local notebook server and opens a browser window. In the following, we use Jupyter notebooks which are very popular in the Python community. Alternatively, a legacy SageMath notebook server can be started with `sage -notebook=sagenb`.

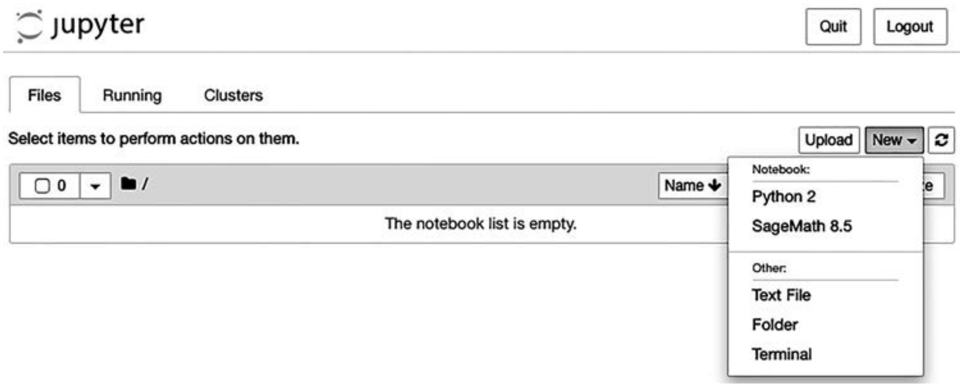


Figure 0.1. Creating a new browser notebook.

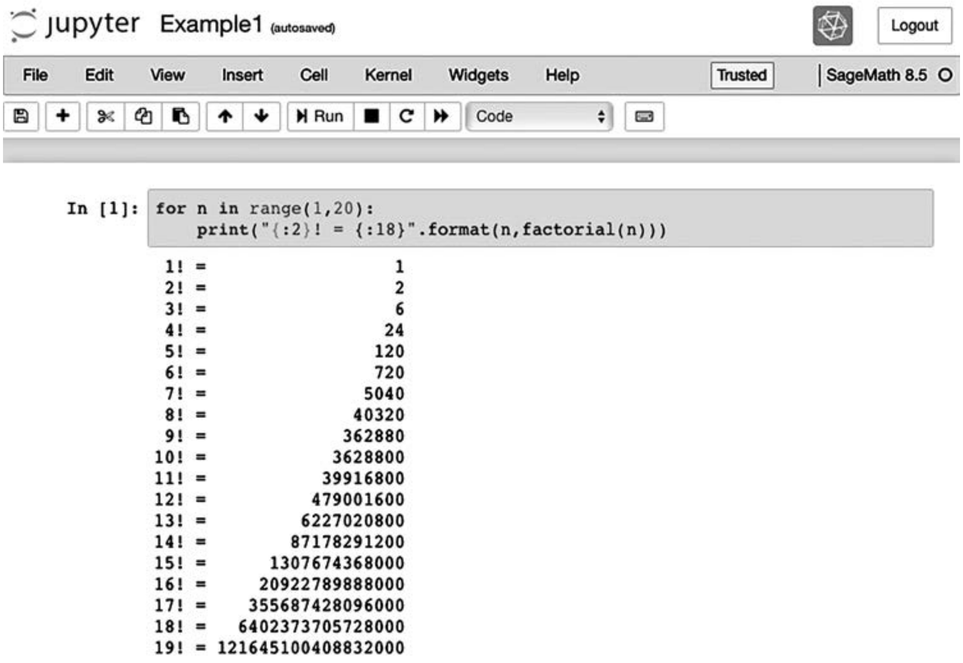
A new notebook is created by clicking on the 'New' button in the upper right corner and choosing SageMath (see Figure 0.1). The commands and the code are written in *input cells* and a cell is evaluated by pressing 'Shift + Enter' or by clicking on the *play* symbol in the toolbar. The 'Enter' key does not interpret the code, but rather creates a new line. It is a good practice not to write too much into a single cell, although a cell can contain several lines as well as multiple commands in one line (separated by a semicolon). Do not forget to rename an untitled notebook and to save (and checkpoint) your work via the 'File' menu.

The code shown in Figure 0.2 implements a loop. For each $1 \leq n < 20$ the factorial $n!$ is printed out using the Python format specification. You may be unfamiliar with the way Python structures the code, since it differs from languages like C and Java. The colon in the first line and the indentation of the second line are very important.

0.4. Computations with SageMath

SageMath can perform all kinds of computations, but its symbolic and algebraic capabilities are particularly useful for cryptography. SageMath can work with variables: x is predefined and other variables can be declared by `var('...')`. Some constructions, for example `PolynomialRing`, name the variable used. The following example shows a computation in the polynomial ring $S = \mathbb{Z}[x]$.

```
sage: S.<x> = PolynomialRing(ZZ)
sage: (1+x)^10
x^10 + 10*x^9 + 45*x^8 + 120*x^7 + 210*x^6 + 252*x^5 +
210*x^4 + 120*x^3 + 45*x^2 + 10*x + 1
```



```
In [1]: for n in range(1,20):
            print("{:2}! = {:18}".format(n,factorial(n)))

         1! =                  1
         2! =                  2
         3! =                  6
         4! =                 24
         5! =                120
         6! =                720
         7! =               5040
         8! =              40320
         9! =             362880
        10! =            3628800
        11! =           39916800
        12! =          479001600
        13! =         6227020800
        14! =        87178291200
        15! =      1307674368000
        16! =     20922789888000
        17! =    355687428096000
        18! =   6402373705728000
        19! = 121645100408832000
```

Figure 0.2. SageMath code to print out the factorials.

Now we perform the same computation in the polynomial ring $R = GF(2)[t]$ over the binary field $GF(2)$. The reader is advised to refer to Chapters 3 and 1 for the mathematical background of the following examples.

```
sage: R.<t> = PolynomialRing(GF(2))
sage: (1+t)^10
t^10 + t^8 + t^2 + 1
```

We can also construct the residue class ring

$$F = R/(t^8 + t^4 + t^3 + t + 1) = GF(2)[t]/(t^8 + t^4 + t^3 + t + 1)$$

which defines the field $GF(256)$ with $2^8 = 256$ elements. The variable a represents the residue class of t modulo $t^8 + t^4 + t^3 + t + 1$. All nonzero elements in $GF(256)$ are invertible and we compute the multiplicative inverse of $a + 1$.

```
sage: F.<a>=R.quotient_ring(t^8+t^4+t^3+t+1)
sage: 1/(a+1)
a^7 + a^6 + a^5 + a^4 + a^2 + a
```

We verify the result by multiplying the residue classes. Note the difference to the multiplication in $R = GF(2)[t]$.

```
sage: (a+1)*(a^7 + a^6 + a^5 + a^4 + a^2 + a)
1
sage: (t+1)*(t^7 + t^6 + t^5 + t^4 + t^2 + t)
t^8 + t^4 + t^3 + t
```

We define a 4×4 matrix over $GF(256)$ and let SageMath compute the inverse:

```
sage: M=matrix(F,[[a,a+1,1,1],[1,a,a+1,1],[1,1,a,a+1],
[a+1,1,1,a]])
sage: 1/M
[a^3 + a^2 + a     a^3 + a + 1  a^3 + a^2 + 1         a^3 + 1]
[      a^3 + 1  a^3 + a^2 + a     a^3 + a + 1  a^3 + a^2 + 1]
[a^3 + a^2 + 1         a^3 + 1  a^3 + a^2 + a     a^3 + a + 1]
[  a^3 + a + 1  a^3 + a^2 + 1         a^3 + 1  a^3 + a^2 + a]
```

In Chapter 5, we will see that the field $GF(256)$ and the matrix M are used in the AES block cipher.

Chapter 1

Fundamentals

Modern cryptography relies on mathematical structures and methods, and this chapter contains the mathematical background from discrete mathematics, computational complexity and probability theory. We recapitulate elementary structures like sets, relations, equivalence classes and functions in Section 1.1. Fundamental combinatorial facts are outlined in Section 1.2 and the asymptotic notation is explained. Section 1.3 discusses complexity and the Big-O notation. Section 1.4 then deals with basic probability theory. Random numbers and the birthday problem are addressed in Section 1.5.

For a general introduction to undergraduate mathematics the reader may, for example, refer to the textbook [**WJW$^+$14**]. Discrete mathematics and its applications are discussed in [**Ros12**].

1.1. Sets, Relations and Functions

The most elementary mathematical structure is sets.

Definition 1.1 (Cantor's definition of sets)**.** A *set* M is a well-defined collection of *distinct objects*. If the object x is a member of M, write $x \in M$ or otherwise $x \notin M$. One writes $N \subset M$ if N is a subset of M, i.e., if all elements of N are also elements of M.

Example 1.2. Basic examples of finite sets are $\{\ \} = \emptyset$ (the empty set), $\{0\}$ (a set with one element), $\{0, 1\}$ (the binary numbers) and $\{A, B, C, \dots, Z\}$ (the set of capital letters). The standard sets of numbers $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ (positive integers, integers, rational numbers, real and complex numbers) are examples of infinite sets.

Definition 1.3. The *cardinality* or *size* of a finite set X is the number of its elements and is denoted by $|X|$. $\Diamond$

Note that there are other notions of *size* (see Warning 1.34 below): the size of an integer is the *number of bits* needed to represent it and the size of a binary string is its *length.*

Sets can be defined *explicitly*, for example by enumeration or by intervals of real numbers, or *implicitly* by formulas.

Example 1.4. $M = \{x \in \mathbb{Z} \mid x^4 < 50\}$ implicitly describes the set of integers x for which $x^4 < 50$ holds. This set can also be described explicitly:

$$M = \{-2,\ -1,\ 0,\ 1,\ 2\}. \qquad \diamondsuit$$

There are several elementary set operations: $M \cup N$ (union), $M \cap N$ (intersection), $M \setminus N$ (set difference), $M \times N$ (Cartesian product), M^n (n-ary product) and $\mathcal{P}(M)$ (power set).

Example 1.5. $M = \{0,1\}^{128}$ is the set of binary strings of length 128. Elements in M can be written in the form $b_1 b_2 \dots b_{128}$ or $(b_1,\ b_2,\ \dots,\ b_{128})$ in vectorial notation. An element of M could, for example, represent one block of plaintext or ciphertext data. The cardinality of M is very large:

$$|M| = 2^{128} \approx 3.4 \cdot 10^{38}$$

```
sage: 2^128
340282366920938463463374607431768211456
```

Remark 1.6. It is useful to help understand the difference between small, big and inaccessible numbers in practical computations. For example, one can easily store one terabyte (10^{12} bytes, i.e., around 2^{43} bits) of data. On the other hand, a large amount of resources are required to store one exabyte (one million terabytes) or 2^{63} bits, and more than 2^{100} bits are out of reach.

The number of computing steps is also bounded: less than 2^{40} steps (say CPU clocks) are easily possible, 2^{60} operations require a lot of computing resources and take a significant amount of time, and more than 2^{100} operations are unfeasible. It is for example impossible to test 2^{128} different keys with conventional (non-quantum) computers.

Definition 1.7. A *function*, *mapping* or *map* $f : X \to Y$ consists of two sets (the *domain* X and the *codomain* Y) and a rule which assigns an output element (an *image*) $y = f(x) \in Y$ to each input element $x \in X$. The set of all $f(x)$ is a subset of Y called the *range* or the *image* $im(f)$. Any $x \in X$ with $f(x) = y$ is called a *preimage* of y. Let $B \subset Y$; then we say that $f^{-1}(B) = \{x \in X \mid f(x) \in B\}$ is the *preimage* or *inverse image* of B under f.

Example 1.8. Let $f : \{0,1\}^4 \to \{0,1\}$ be defined by

$$f(b_1, b_2, b_3, b_4) = b_1 \oplus b_2 \oplus (b_3 \cdot b_4) = b_1 \oplus b_2 \oplus b_3 b_4.$$

Refer to Table 1.1 for the definition of XOR ($\oplus$) and AND ($\cdot$). For example, $(1,1,1,1)$ is a preimage of 1 and $(0,1,0,0)$ is another preimage of 1. $(0,0,0,0)$ is a preimage of 0 and the image of f is $im(f) = \{0,1\}$. The function f is surjective, but not injective (see Definition 1.10 below).

Table 1.1. XOR and AND operations.

$\oplus$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Warning 1.9. A mathematical function is not the same as an *algorithm*. An algorithm is a step-by-step and possibly recursive procedure which produces output from given input. There can be different algorithms (or no known algorithm at all) for a given mathematical function. Furthermore, algorithms can also use random data as input and can have different behaviors on different runs. In computer science, a function is a piece of code in some programming language that performs a specific task.

Every set X has an *identity function* $id_X : X \to X$, which maps each $x \in X$ to itself. Functions can be *composed* if the range of the first function lies within the domain of the second function. Let $f : X \to Y$ and $g : Y \to Z$ be functions. Then there is a composite function $g \circ f : X \to Z$ with $(g \circ f)(x) = g(f(x))$ (see Figure 1.1).

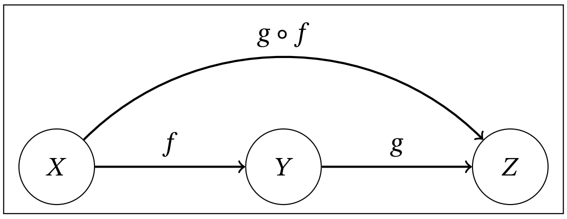

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] & Y \arrow[r, "g"] & Z \end{tikzcd}$$

Figure 1.1. Composition of functions.

Definition 1.10. Let $f : X \to Y$ be a function.

- f is *injective* if different elements of the domain map to different elements of the range: for all $x_1,\ x_2 \in X$ with $x_1 \neq x_2$ we have $f(x_1) \neq f(x_2)$. Equivalently, f is injective if for all $x_1,\ x_2 \in X$:

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2.$$

- f is *surjective* or *onto* if every element of the codomain Y is contained in the image of f, i.e., for every $y \in Y$ there exists an $x \in X$ (a *preimage*) with $f(x) = y$. In other words, f is surjective if
$$im(f) = Y.$$
- f is *bijective* if it is both injective and surjective. Bijective functions possess an inverse map $f^{-1} : Y \to X$ such that $f^{-1} \circ f = id_X$ and $f \circ f^{-1} = id_Y$ (see Figure 1.2). Such functions are also called *invertible*.

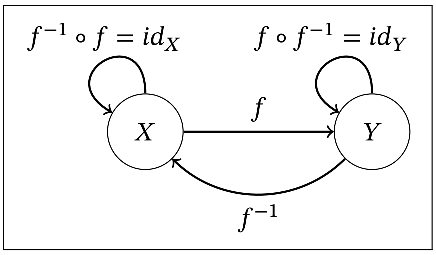


Figure 1.2. Inverse map and its properties.

Example 1.11. Let $f : \{0,1\}^2 \to \{0,1\}^2$ be defined by
$$f(b_1, b_2) = (b_1 \oplus b_2,\, b_1).$$
Since $f(0,0) = (0,0)$, $f(0,1) = (1,0)$, $f(1,0) = (1,1)$ and $f(1,1) = (0,1)$, the map is bijective. The inverse map $f^{-1} : \{0,1\}^2 \to \{0,1\}^2$ is given by
$$f^{-1}(b_1, b_2) = (b_2,\, b_1 \oplus b_2).$$
One has $f^{-1}(0,0) = (0,0)$, $f^{-1}(0,1) = (1,1)$, $f^{-1}(1,0) = (0,1)$ and $f^{-1}(1,1) = (1,0)$. $\diamondsuit$

Invertible (bijective) maps play an important role in ciphering. An encryption map E must have a corresponding decryption map D such that $D \circ E = id$ (the identity map on the plaintext space). Encryption maps are often bijective, although a slightly weaker property is sufficient (E has to be injective and D must be surjective). Ciphering is often defined as a composition of several operations. These operations are not necessarily bijective, even if their composition is ultimately bijective. On the other hand, a composition of bijective maps is of course bijective.

Lemma 1.12. *Let $f : X \to Y$ be a map between finite sets; then:*

(1) *If f is injective then $|X| \leq |Y|$.*
(2) *If f is surjective then $|X| \geq |Y|$.*
(3) *If f is bijective then $|X| = |Y|$.* $\diamondsuit$

Note that the above conditions are only necessary and not sufficient.

Remark 1.13. The contraposition of Lemma 1.12 (1) is called the *pigeonhole principle*: if $|X| > |Y|$ then f is not injective. Suppose X is a set of pigeons and Y a set of holes. If there are more pigeons than holes, then one hole has more than one pigeon.

Definition 1.14. A set S is said to be *countably infinite* if there is a bijective map

$$f : S \to \mathbb{N}$$

from S to the set of natural numbers. We say that a set S is *countable* if it is finite or countably infinite.

Example 1.15. $\mathbb{N}$ and $\mathbb{Z}$ are countably infinite sets. The sets $\mathbb{Z}^n$ (for $n \in \mathbb{N}$) and $\mathbb{Q}$ are also countable (see Exercise 3). However, a famous result of Cantor says that the set $\mathbb{R}$ of all real numbers is *uncountable*. $\Diamond$

The *floor*, *ceiling* and *rounding* functions are often used in numerical computations.

Definition 1.16. Let $x \in \mathbb{R}$.

(1) $\lfloor x \rfloor$ is the greatest integer less than or equal to x.

(2) $\lceil x \rceil$ is the least integer greater than or equal to x.

(3) $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$ is rounding x to the nearest integer (round half up).

Example 1.17. $\lfloor 1.7 \rfloor = 1$, $\lceil -2.4 \rceil = -2$ and $\lfloor 1.5 \rceil = 2$. $\Diamond$

Functions from X to Y are particular cases of *relations between X and Y*:

Definition 1.18. A binary relation (or *relation*) between X and Y is a subset

$$R \subset X \times Y.$$

Hence a relation is a set of pairs (x, y) with $x \in X$ and $y \in Y$. If $X = Y$ then R is called a (binary) relation on X. $\Diamond$

A function $f : X \to Y$ defines a relation between X and Y:

$$R = \{(x, y) \in X \times Y \mid y = f(x)\}.$$

R contains all tuples $(x, f(x))$ with $x \in X$. This relation is also called the *graph* of f. However, a relation R between X and Y does not necessarily define a function. A relation R defines a function if and only if each $x \in X$ occurs exactly once as a first component in the relation R.

Equivalence relations on a set X are of special importance. An equivalence relation induces a segmentation of a set into disjoint subsets. The collection of these subsets defines a new set and elements in the same subset are said to be *equivalent*.

Definition 1.19. Let R be a relation on X. Then R is called an *equivalence relation* if it satisfies the following conditions:

(1) R is reflexive, i.e., $(x,x) \in R$ for all $x \in X$, and

(2) R is symmetric, i.e., if $(x,y) \in R$ then $(y,x) \in R$, and

(3) R is transitive, i.e., if $(x,y) \in R$ and $(y,z) \in R$ then $(x,z) \in R$. ◊

If R is an equivalence relation and $(x,y) \in R$, then x and y are called equivalent and we write $x \sim y$. For $x \in X$, the subset $\overline{x} = \{y \in X \mid x \sim y\} \subset X$ is called the *equivalence class* of x and all elements in $\overline{x}$ are representatives of that class. The above conditions of an equivalence relation ensure that the equivalence classes are disjoint and their union is X. The set of equivalence classes is said to be the *quotient set* $X/\sim$.

Note: the *elements* of $X/\sim$ are *sets*. By abuse of notation, the same x is sometimes viewed as an element of X and of $X/\sim$.

Example 1.20. We define an equivalence relation R_2 on $X = \mathbb{Z}$:

$$R_2 = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x - y \in 2\mathbb{Z}\}.$$

Pairs $(x,y) \in R_2$ have the property that their difference $x - y$ is even, i.e., divisible by 2. For example, the tuples $(2,4)$, $(3,1)$ and $(-1,5)$ are all elements of R_2, but $(0,1) \notin R_2$ and $(-3,4) \notin R_2$. One can easily check that R_2 is an equivalence relation. The equivalence class of $x \in \mathbb{Z}$ is

$$\overline{x} = \{\dots,\ x-4,\ x-2,\ x,\ x+2,\ x+4,\dots\}.$$

There are only two *different* equivalence classes:

$$\overline{0} = \{\dots,-4,-2,0,2,4,\dots\} \text{ and } \overline{1} = \{\dots,-3,-1,1,3,5,\dots\}.$$

For example, one has $\overline{-2} = \overline{0} = \overline{2}$ and $\overline{-1} = \overline{1} = \overline{3}$. Hence the quotient set $\mathbb{Z}/\sim$ only has two elements. This set is denoted by $\mathbb{Z}_2$, $\mathbb{Z}/(2)$, $GF(2)$ or $\mathbb{F}_2$, and we call it the field of *residue classes modulo* 2. It can be identified with the binary set $\{0,1\}$. ◊

A generalization of the above example yields the *residue classes* modulo n:

Example 1.21. Let $n \in \mathbb{N}$ and $n \geq 2$. Consider the following equivalence relation R_n on $X = \mathbb{Z}$:

$$R_n = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x - y \in n\mathbb{Z}\}.$$

Pairs $(x,y) \in R_n$ have the property that their difference $x-y$ is divisible by n, i.e., $x-y$ is a multiple of n. For example, let $n = 11$. Then the tuples $(2,13)$ and $(-1,10)$ are elements of R_{11}, but $(0,10) \notin R_{11}$ and $(-2,13) \notin R_{11}$.

Similar to the example above, R_n is an equivalence relation. The equivalence class of $x \in \mathbb{Z}$ is the set

$$\overline{x} = \{\dots,\ x-2n,\ x-n,\ x,\ x+n,\ x+2n,\ \dots\}.$$

Now we have n *different* equivalence classes and the quotient set $\mathbb{Z}/\sim$ has n elements. We call this set the *residue classes modulo n* or *integers modulo n* and denote it by $\mathbb{Z}_n$ or $\mathbb{Z}/(n)$. Each residue class has a *standard representative* in the set $\{0, 1, \dots, n-1\}$ and elements in the same residue class are called *congruent modulo n*. ◊

Two integers are congruent modulo n if they have the same remainder when they are divided by n. In many programming languages, the remainder of the integer division $a : n$ is computed by $a \% n$, but note that the result may be negative for $a < 0$, whereas the standard representative of a modulo n is non-negative.

Example 1.22. Let $n = 11$. Then $\mathbb{Z}_{11} = \{\overline{0}, \overline{1}, \dots, \overline{10}\}$ has 11 elements. One has $\overline{-14} = \overline{8}$ since $-14-8 = -22$ is a multiple of 11. The integers 8 and -14 are congruent modulo 11 and one writes $-14 \equiv 8 \mod 11$. The standard representative of this residue class is 8, and $-3, -14, \dots$ as well as $19, 30, \dots$ are other representatives of the same residue class. Here is an example using SageMath:

```
sage: mod(-892342322327,11)
6
```

The discussion of residue classes will be continued in Section 3.2.

Definition 1.23. A map $f : \{0,1\}^n \to \{0,1\}$ is called an n-variable *Boolean function* and a map $f : \{0,1\}^n \to \{0,1\}^m$ is called an (n,m)-vectorial Boolean function. A vectorial Boolean function can be written as an m-tuple of n-variable Boolean functions: $f = (f_1, f_2, \dots, f_m)$. ◊

Boolean functions can be represented by their *truth table*. Since the table of an n-variable Boolean function has 2^n entries, this is only reasonable for small n. Another important representation is the *algebraic normal form* (ANF). This form uses XOR ($\oplus$) and AND ($\cdot$) combinations of the binary variables.

An n-variable Boolean function has a unique representation as a polynomial in n variables, say $x_1, x_2, \dots, x_n$:

$$f(x_1, x_2, \dots, x_n) = \bigoplus_{I \subset \{1,\dots,n\}} a_I \cdot \prod_{i \in I} x_i.$$

The coefficients a_I are either 0 or 1 and f is a sum (XOR) of products (AND) of the variables, for example

$$f(x_1, x_2, x_3) = x_1 \oplus x_1 x_2 \oplus x_2 x_3 \oplus x_1 x_2 x_3.$$

One can view f as a polynomial in n variables with coefficients in $\mathbb{Z}_2$ (residue classes modulo 2) and write $+$ instead of $\oplus$, e.g.,

$$f(x_1, x_2, x_3) = x_1 + x_1 x_2 + x_2 x_3 + x_1 x_2 x_3 \mod 2.$$

The *algebraic degree* of f is the maximal length of the products which appear (with nonzero coefficient) in the above representation. If f is a constant function, then the degree is 0. In the above example, the degree is 3.

We note that higher powers of a variable x_i are not needed since $x_i^k = x_i$ for all $k \geq 1$ and $x_i \in \{0,1\}$. Boolean functions of degree ≤ 1 are called *affine*. If the degree is ≤ 1 and the constant part is 0, then the function is *linear*. An n-variable linear Boolean function is a linear mapping from $GF(2)^n$ to $GF(2)$. Linear maps are discussed in Section 4.4.

The *degree* of a vectorial Boolean function is the maximal degree of its component functions. A vectorial Boolean function is called affine if all component functions are affine.

Example 1.24. (1) $f(x_1,x_2,x_3) = x_1x_2+x_2x_3+x_1+1 \mod 2$ is a 3-variable Boolean function of algebraic degree 2.

(2) $f(x_1,x_2,x_3,x_4) = (x_3,\ x_1+x_2,\ x_1+x_4) \mod 2$ is a $(4,3)$-vectorial Boolean function. The algebraic degree is 1 and the function is linear since all constants are 0.

(3) Let $f = f(x_0,x_1)$ be a 2-variable Boolean function given by the following table:

x_1	x_0	$f(x)$
0	0	1
0	1	1
1	0	0
1	1	1

We use SageMath to determine the algebraic normal form:

```
sage: B = BooleanFunction([1,1,0,1])
sage: B.algebraic_normal_form()
x0*x1 + x1 + 1
```

We find that $f(x_0,x_1) = x_0x_1 + x_1 + 1 \mod 2$. The algebraic degree is 2 and f is neither linear nor affine.

1.2. Combinatorics

Combinatorics investigates finite or countable discrete structures. We are interested in properties of finite sets like $\{1,2,\ldots,n\}$ or $\{0,1\}^n$ which often occur in cryptographic operations.

Definition 1.25. The *factorial* of a non-negative integer n is defined as follows:

$$n! = \begin{cases} 1 & \text{for } n = 0, \\ 1 \cdot 2 \cdots n & \text{for } n \geq 1. \end{cases}$$

The *binomial coefficients* have the following formula:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \text{ for integers } 0 \leq k \leq n\,. \qquad \diamondsuit$$

The factorial increases very fast, for example

$$25! = 15511210043330985984000000 \text{ and } 50! \approx 3.04 \cdot 10^{64}.$$

An efficient way to compute the binomial coefficients is the reduced formula

$$\binom{n}{k} = \binom{n}{n-k} = \frac{n(n-1)\cdots(n-(k-1))}{k!}\,.$$

Example 1.26. One has $\binom{15}{8} = \binom{15}{7} = \frac{15\cdot14\cdot13\cdot12\cdot11\cdot10\cdot9}{7!} = \frac{32432400}{5040} = 6435$. The following SageMath code computes all binomials $\binom{15}{n}$ for $0 \leq n \leq 15$:

```
sage: for n in range(0,16):
          print binomial(15,n),
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
```

The binomial coefficients appear for example in the *number of subsets* of a given finite set and in expansions of terms like $(a+b)^n$.

Proposition 1.27. *Let X and Y be finite sets. Then*

(1) $|X \times Y| = |X| \cdot |Y|$ *and* $|X^k| = |X|^k$ *for* $k \in \mathbb{N}$.

(2) *Let* $n = |X|$ *be the cardinality of X and* $k \leq n$. *Then the number of subsets of X of cardinality k is* $\binom{n}{k}$. ♢

Note that there is a difference between k-tuples and subsets of cardinality k. In the first case, the order of elements is important and in the second case it is not.

Example 1.28. There are $\binom{128}{2} = \frac{128\cdot127}{2} = 8128$ different binary words of length 128 with exactly two ones and 126 zeros. Indeed, each subset of $X = \{1, 2, \ldots, 128\}$ with two elements gives the two positions where the digit is equal to 1. ♢

Permutations of finite sets are of great importance in cryptography.

Definition 1.29. Let S be a finite set. A *permutation* σ of S is a bijective map $\sigma : S \to S$. ♢

Permutations of a finite set S can be written using a two-line matrix notation. The first row lists the elements of S and the image of each element is given below in the second row. The first row can be omitted if S is given and the elements are naturally ordered.

Example 1.30. Let $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$; then the following row describes a permutation of S:

$$(5\ 7\ 1\ 2\ 8\ 6\ 3\ 4).$$

Proposition 1.31. *Let S be a finite set and $|S| = n$; then there are $n!$ permutations of S.*

Proof. Write $S = \{x_1, x_2, \dots, x_n\}$ and let σ be a permutation of S. There are n different choices for $\sigma(x_1)$. Since σ is injective, there remain $n - 1$ possible images for $\sigma(x_2)$, then $n-2$ possibilities for $\sigma(x_3)$, etc. The last image $\sigma(x_n)$ is fixed by the other choices. Hence one has $n(n-1)(n-2)\cdots 1 = n!$ different permutations for $\sigma : S \to S$. □

Cryptographic operations are often based on permutations. Indeed, a randomly chosen family of permutations of a set like $M = \{0, 1\}^{128}$ would constitute an ideal block cipher. Unfortunately, it is impossible to write down or to store a general permutation since M has 2^{128} elements (see Example 1.5). Much simpler are *linear maps* (see Section 4.4) and in particular *bit permutations*, which permute only the *position* of the bits. For example, we might take the permutation (5 7 1 2 8 6 3 4) of $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$ in Example 1.30 to define a permutation on the set $X = \{0, 1\}^8$. A byte $(b_1, b_2, \dots, b_8) \in X$ is mapped to $(b_5, b_7, b_1, b_2, b_8, b_6, b_3, b_4) \in X$. Such bit permutations are used within ciphering operations, but one has to take into account the fact that bit permutations are (in contrast to general permutations) *linear* maps.

1.3. Computational Complexity

One often needs to analyze the computational complexity of algorithms, i.e., the necessary time and space relative to the input size. The precise values, for example the number of steps or CPU-clocks and the required memory in bytes, depend on the platform, the implementation and the chosen measurement units, but the *growth* of time or space (in terms of the input size) is often independent of the operational environment and the parameters chosen. The *Big-O* notation describes the growth of functions and is used to estimate the complexity of algorithms in terms of the input size.

Definition 1.32 (Big-O notation). Let $f, g : \mathbb{N} \to \mathbb{R}$ be two functions on $\mathbb{N}$. Then we say that g is an *asymptotic upper bound* for f if there exists a real number $C \in \mathbb{R}$ and an integer $n_0 \in \mathbb{N}$ such that

$$|f(n)| \leq C\,|g(n)| \text{ for all } n \geq n_0,$$

and one writes $f = O(g)$ or $f \in O(g)$. ♢

An asymptotic upper bound $g(n)$ is usually chosen to be as simple and as small as possible. We say that f has linear, quadratic, cubic or polynomial growth in n if $f = O(n)$, $f = O(n^2)$, $f = O(n^3)$ or $f = O(n^k)$ for some $k \in \mathbb{N}$, respectively.

Example 1.33. (1) Let $f(n) = 2n^3+n^2+7n+2$. Since $n^2 \leq n^3$, $n \leq n^3$ and $1 \leq n^3$ for $n \geq 1$, one has $f(n) \leq (2+1+7+2)n^3$. Set $C = 12$ and $n_0 = 1$. Thus $f = O(n^3)$ and f has cubic growth in n.

(2) Let $f(n) = 100 + \frac{20}{n+1}$. Set $C = 101$ and $n_0 = 19$. Since $\frac{20}{n+1} \leq 1$ for $n \geq 19$, we have $f = O(1)$. Hence f is asymptotically bounded by a constant.

(3) Let $f(n) = 5\sqrt{2^{n+3} + n^2 - 2n}$; then $f = O(2^{n/2})$ so that f grows exponentially in n. ♢

The Big-O notation is often used to assess the running time of an algorithm in a worst-case scenario. An asymptotic upper bound does not depend on the measuring unit or the platform, since the values would differ only by a multiplicative constant. If the running time function $f(n)$ has polynomial growth in the input length n, i.e., $f = O(n^k)$, then $f(n)$ is bounded by Cn^k for constants C, k and large n. For example, if n is doubled, then the upper bound is multiplied only by the constant 2^k.

Algorithms with *polynomial* running time in terms of the input size are considered to be *efficient* or *fast*. Many standard algorithms, for example adding or multiplying two numbers, are polynomial. On the other hand, an algorithm that loops over every instance of a set with 2^n elements is *exponential* in n. A problem is called *hard* if no efficient, i.e., polynomial-time, algorithm exists that solves the problem.

A *decision problem* has only two possible answers, yes or no. Decision problems for which a polynomial time algorithm exists are said to belong to the *complexity class* **P**. The class **NP** (nondeterministic polynomial) is the set of decision problems which can be *verified* in polynomial-time. Checking the correctness of a proof is apparently easier than solving a problem. Whether the class **NP** is strictly larger than **P** is a major unresolved problem in computer science.

In computer science, one is usually interested in the *worst-case* complexity of an algorithm which solves a certain problem. However, the worst-case complexity is hardly relevant for attacks against cryptographic systems. A cryptosystem is certainly insecure if attacks are inefficient in certain bad cases but efficient in other cases. Instead, the *average-case* complexity of algorithms that break a scheme should be large. A secure scheme should provide protection in *almost all cases* and the probability that a polynomial time algorithm breaks the cryptosystem should be very small.

Note that there is not only the time complexity but also the *space complexity* of an algorithm. The space complexity measures the memory or storage requirements in terms of the input size.

Warning 1.34. The complexity is measured in terms of the input *size*, not the input value! The following formula gives the relation between a positive integer n and its size:

$$\text{size}(n) = \lfloor \log_2(n) \rfloor + 1.$$

The size of an integer is the number of bits that is needed to represent it. For a given binary string m, the bit-length is also called size and denoted by $|m|$.

In cryptographic applications, the key size is usually bounded by several hundred or thousand bits, but the numbers represented can be very large.

Example 1.35. Let $n = 10^{24}$. Then $\text{size}(n) = \lfloor 24 \cdot \log_2(10) \rfloor + 1 = 80$. Hence 80 bits are needed for the binary representation of 10^{24}.

Example 1.36. Let $n \in \mathbb{N}$ and $1^n = 1 \dots 1$ be the *unary string* of n ones. Obviously, $\text{size}(1^n) = n$. An algorithm that takes 1^n as input is polynomial if the running time is polynomial in n. Of course, 1^n is a very inefficient way to represent a positive integer n, but it allows us to set the input *size* to n. This will be used in Chapter 2, where algorithms are often expected to be polynomial in a security parameter n, for example the key length.

Example 1.37. Suppose we want to compute the prime divisors of an integer n. A related decision problem is to determine whether or not n is a prime number.

Our algorithm uses trial divisions by all odd positive integers $\leq \sqrt{n}$. The number of divisions (i.e., a multiple of the running time) is at most $\lfloor \frac{1}{2}\sqrt{n} \rfloor$. Since $\sqrt{n} = 2^{\frac{1}{2}\log_2 n}$, the worst-case running time function is $O(2^{\frac{1}{2}\,\text{size}(n)})$ and therefore *exponential*. On the other hand, our algorithm has polynomial space complexity, since we only need to store n and a small number of auxiliary variables. Note that the factor $\frac{1}{2}$ in the *exponent* of the running time function must not be omitted in the Big-O notation! ♢

One should understand the limitations of the asymptotic notation. First, the upper bound applies only if n is larger than some unknown initial value n_0. Furthermore, the multiplicative constant C can be large. For a given input value, an algorithm with polynomial running time may even be slower than an exponential-time algorithm! However, for large n the polynomial-time algorithm will eventually be faster.

Bounded functions are of type $O(1)$ and functions of type $O(\frac{1}{n})$ converge to 0. *Negligible* functions approach zero faster than $\frac{1}{n}$ and any other inverse polynomial:

Definition 1.38. Let $f : \mathbb{N} \to \mathbb{R}$ be a function. One says that f is *negligible* in n if $f = O(\frac{1}{q(n)})$ for all polynomials q, or equivalently if $f = O(\frac{1}{n^c})$ for all $c > 0$. ♢

Hence negligible functions are eventually smaller than any inverse polynomial. This means that $f(n)$ approaches zero faster than any of the functions $\frac{1}{n}, \frac{1}{n^2}, \frac{1}{n^3}$, etc.

Example 1.39. $f(n) = 10e^{-n}$ and $2^{-\sqrt{n}}$ are negligible in n, whereas $f(n) = \frac{1}{n^2+3n}$ is not negligible since $f(n) = O(\frac{1}{n^2})$, but $f \neq O(\frac{1}{n^3})$. ♢

Finally, we define the *soft-O* notation which ignores logarithmic factors.

Definition 1.40. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. Then $f = \tilde{O}(g)$ if there exists $k \in \mathbb{N}$ such that $f = O(\ln(n)^k g(n))$.

Example 1.41. (1) Let $f_1(n) = \log_2(n)$. Then $f_1 = \tilde{O}(1)$.
(2) Let $f_2(n) = \log_{10}(n)^3 n^2$. Then $f_2 = \tilde{O}(n^2)$.

1.4. Discrete Probability

Probability theory is an important foundation of modern cryptography. We only consider *discrete probability spaces*.

Definition 1.42. Let Ω be a countable set, $\mathcal{S} = \mathcal{P}(\Omega)$ the power set of Ω and $Pr : \mathcal{S} \to [0, 1]$ a function with the following properties:

(1) $Pr[\Omega] = 1$.
(2) If $A_1, A_2, A_3, \dots$ are pairwise disjoint subsets of A, then

$$Pr\left[\bigcup_i A_i\right] = \sum_i Pr[A_i].$$

The triple $(\Omega, \mathcal{S}, Pr)$ is called a *discrete probability space*. Ω is called the *sample space* and we say that Pr is a *discrete probability distribution* on Ω. The subsets $A \subset \Omega$ are said to be *events* and $Pr[A]$ is the *probability* of A. If Ω is finite, then $(\Omega, \mathcal{S}, Pr)$ is called a *finite probability space*. ♢

Note that the family of sets in (2) is either finite or countably infinite. Since all probabilities are non-negative and the sum is bounded by 1, the series converges and is also invariant under a reordering of terms.

Remark 1.43. In measure theory, a triple $(\Omega, \mathcal{S}, Pr)$, where Pr is a σ-additive function on a set $\mathcal{S} \subset \mathcal{P}(\Omega)$ of *measurable sets* such that $Pr[\Omega] = 1$, is called a *probability space*. We only consider the case of a *countable* sample space Ω and assume that all subsets (events) are measurable, i.e., $\mathcal{S} = \mathcal{P}(\Omega)$. A discrete probability distribution is fully determined by the values on the singletons $\{\omega\}$ (the elementary events). We define the function

$$p(\omega) = Pr[\{\omega\}]$$

and obtain for all events $A \subset \Omega$:

$$Pr[A] = \sum_{\omega \in A} p(\omega).$$

Hence a discrete probability distribution is fully determined by a function $p : \Omega \to [0,1]$ with $\sum_{\omega \in \Omega} p(\omega) = 1$.

Example 1.44. Let $\Omega = \mathbb{N}$ and $0 < p < 1$. Define a discrete probability distribution *Pr* on $\mathbb{N}$ by

$$p(k) = Pr[\{k\}] = (1-p)^{k-1} p.$$

The formula for the geometric series implies that $Pr[\mathbb{N}] = 1$:

$$\sum_{k=1}^{\infty} p(k) = p \sum_{k=1}^{\infty} (1-p)^{k-1} = p \frac{1}{1-(1-p)} = 1.$$

Pr is called a *geometric distribution*. Suppose the probability of success in an experiment is equal to p. Then the geometric distribution gives the probability that the first success requires k independent trials.

Definition 1.45. Let *Pr* be a probability distribution on a finite sample space Ω. *Pr* is a *uniform distribution* if all elementary events have equal probability, i.e., if

$$p(\omega) = Pr[\{\omega\}] = \frac{1}{|\Omega|} \text{ for all } \omega \in \Omega. \qquad \diamond$$

Examples of uniform distributions include a fair coin or a dice. Random number generators should produce uniformly distributed numbers. A uniform distribution is sometimes assumed when other information is not available. Cryptographic schemes (see Chapter 2) often assume that a key is chosen *uniformly at random* so that all possible keys have the same probability.

Definition 1.46. Let A, B and $A_1, \dots, A_n$ be events in a probability space.

(1) A and B are said to be *independent* if the joint probability equals the product of probabilities:

$$Pr[A \cap B] = Pr[A] \cdot Pr[B].$$

(2) $A_1, A_2, \dots, A_n$ are *mutually independent* if for every non-empty subset of indices $I \subset \{1, \dots, n\}$, one has

$$Pr\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} Pr[A_i]. \qquad \diamond$$

Note that mutual independence of n events is stronger than pairwise independence of all pairs (see Example 1.53 below).

Definition 1.47. Let A and B be two events in a probability space and suppose that

$Pr[A] \neq 0$. Then the *conditional probability* $Pr[B|A]$ of B given A is defined by

$$Pr[B|A] = \frac{Pr[A \cap B]}{Pr[A]}. \qquad \Diamond$$

Obviously, $Pr[B|A] = Pr[B]$ holds if and only if A and B are independent.

The sample space Ω of a probability space is often mapped to a standard space and in particular to the real numbers. Such a map on Ω is called a *random variable*. For example, the outcome of rolling two dice can be mapped to $\{1, 2, \dots, 6\}^2$ or to $\{2, 3, \dots, 12\}$ if the numbers on the dice are added.

Definition 1.48. Let $Pr : \Omega \to [0, 1]$ be a discrete probability distribution; then a function $X : \Omega \to \mathbb{R}$ is called a *real random variable*. For any $x \in \mathbb{R}$, one obtains an event $X^{-1}(x) \subset \Omega$ and the probability $Pr[X = x]$ is defined by $Pr[X^{-1}(x)]$. The function $p_X : \mathbb{R} \to [0, 1]$ defined by $p_X(x) = Pr[X = x]$ is called the *probability mass function* (pmf) of X. Furthermore, the *cumulative distribution function* (cdf) $F : \mathbb{R} \to \mathbb{R}$ is defined by $F(x) = Pr[X \leq x]$. $\Diamond$

Note that X induces a discrete probability distribution Pr_X on the countable subset $X(\Omega) \subset \mathbb{R}$. The difference to the original distribution Pr is that the sample space of Pr_X is now a subset of $\mathbb{R}$. If the sample space Ω is already a subset of $\mathbb{R}$, then X is usually the inclusion map.

Example 1.49. Suppose two dice are rolled and the random variable X gives the sum of numbers on the dice. Then $X^{-1}(2) = \{(1, 1)\}$ and $X^{-1}(3) = \{(1, 2), (2, 1)\}$, so that $p_X(2) = P[X = 2] = \frac{1}{36}$, $p_X(3) = P[X = 3] = \frac{1}{36} + \frac{1}{36} = \frac{1}{18}$.

Furthermore, $F(x) = 0$ for $x < 2$, $F(2) = \frac{1}{36}$, $F(3) = \frac{1}{36} + \frac{1}{18} = \frac{1}{12}$, etc., and $F(x) = 1$ for $x \geq 12$.

Definition 1.50. Let Pr be a discrete probability distribution and $X : \Omega \to \mathbb{R}$ a random variable with countable range $X(\Omega) \subset \mathbb{R}$. One defines the *expected value* (also called *expectation*, *mean* or *average*) $E[X]$ and the *variance* $V[X]$ if the sums given below are either finite or the corresponding series converge absolutely:

$$E[X] = \sum_{x \in X(\Omega)} x \cdot Pr[X = x] = \sum_{x \in X(\Omega)} x \cdot p_X(x),$$

$$V[X] = E[(X - E[X])^2] = E[X^2] - (E[X])^2 = \left(\sum_{x \in X(\Omega)} x^2 \cdot p_X(x) \right) - (E[X])^2.$$

The square root $\sigma = \sqrt{V[X]}$ of the variance is called the *standard deviation*. It measures the quadratic deviation from the mean $E[X]$.

Example 1.51. (1) Let Pr be a uniform distribution on a finite set Ω. Assume that the random variable X maps Ω to the set $\{0,\ 1,\ \dots,\ n - 1\}$. The pmf is $p_X(x) = \frac{1}{n}$

for $x = 0, 1, \dots, n-1$, and zero otherwise. The expectation is

$$E[X] = \sum_{k=0}^{n-1} k \cdot \frac{1}{n} = \frac{n(n-1)}{2} \cdot \frac{1}{n} = \frac{n-1}{2}\,.$$

The variance is

$$\begin{aligned} V[X] &= \left(\sum_{k=0}^{n-1} k^2 \cdot \frac{1}{n}\right) - \left(\frac{n-1}{2}\right)^2 \\ &= \frac{n(n-1)(2n-1)}{6} \cdot \frac{1}{n} - \frac{n^2-2n+1}{4} = \frac{n^2-1}{12}. \end{aligned}$$

(2) Consider the experiment of tossing a perfect coin and letting X be the associated random variable having values 0 and 1. It is hardly surprising that the expected value is $E[X] = \frac{1}{2}$. The variance is $V[X] = \frac{1}{4}$ and the standard deviation is $\sigma = \frac{1}{2}$.

Definition 1.52. Let $X_1, X_2, \dots, X_n$ be random variables on a discrete probability space with probability mass functions $p_{X_1}, p_{X_2}, \dots, p_{X_n}$. The random variables are called *mutually independent* if for any sequence $x_1, x_2, \dots, x_n$ of values

$$Pr[X_1 = x_1 \wedge X_2 = x_2 \wedge \cdots \wedge X_n = x_n] = p_{X_1}(x_1) \cdot p_{X_2}(x_2) \cdots \cdot p_{X_n}(x_n).$$

Example 1.53. Let X_1 and X_2 be two binary random variables (values 0 or 1) that are given by tossing two perfect coins so that X_1 and X_2 are independent. Now set $X_3 = X_1 \oplus X_2$. Then X_1, X_2, X_3 are pairwise independent and each of them has a uniform distribution, but they are not mutually independent. We have

$$Pr[X_1 = 1 \wedge X_2 = 1 \wedge X_3 = 1] = 0,$$

since X_3 must be zero if $X_1 = X_2 = 1$, but

$$Pr[X_1 = 1] \cdot Pr[X_2 = 1] \cdot Pr[X_3 = 1] = \left(\frac{1}{2}\right)^3 = \frac{1}{8}.$$

Example 1.54. Let $\Omega = \{0,1\}^8$ be a space of plaintext and ciphertexts. Suppose the plaintexts X are uniformly distributed. Let $\sigma : \Omega \to \Omega$ be a random *bit permutation* (see Section 1.2). Then the ciphertexts $Y = \sigma(X)$ are also uniformly distributed. X and Y are independent if

$$Pr[X = m \wedge Y = c] = Pr[X = m] \cdot Pr[Y = c]$$

holds for all plaintexts m and ciphertexts c. The right side of the equation gives $\frac{1}{2^8} \cdot \frac{1}{2^8} = \frac{1}{2^{16}}$ for all m and c. If m and c possess a different number of ones, then the left side is 0, because such a combination is impossible for a bit permutation. This shows that X and Y are not independent. Later we will see that bit permutations are not secure, since the ciphertext leaks information about the plaintext.

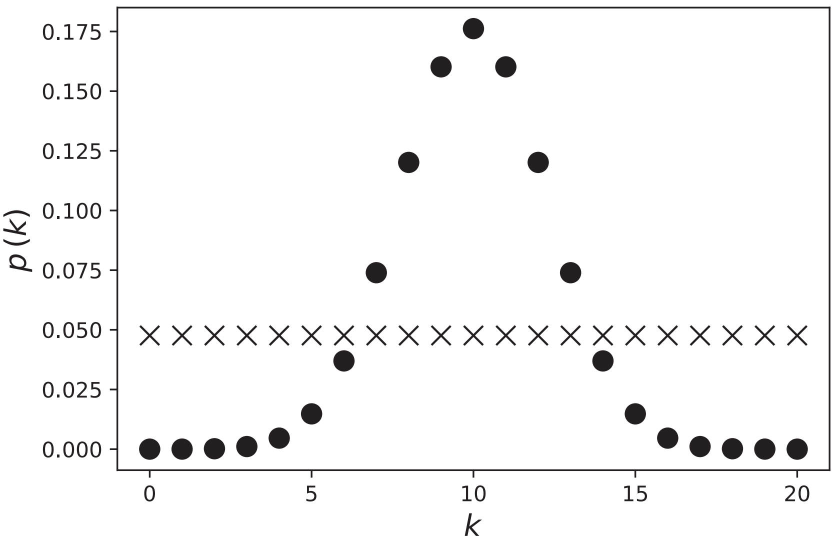


Figure 1.3. Probability mass functions of i) the binomial distribution $B(20, \frac{1}{2})$ (•) and ii) the uniform distribution (×) on $\{0, 1, 2, \ldots, 20\}$.

Example 1.55. Let Pr be a probability distribution on a sample space Ω with two elements. Suppose $X : \Omega \rightarrow \{0,1\}$ is a random variable with $Pr[X = 1] = p$ (success) and $Pr[X = 0] = 1 - p$ (failure). This is called a *Bernoulli trial*. Furthermore, let $X_1, \ldots, X_n$ be n independent identical distributed (i.i.d.) random variables with $X_i = X$, and define

$$Y = X_1 + X_2 + \cdots + X_n.$$

The new random variable Y follows a *binomial distribution* $B(n, p)$ and gives the number of successes in n independent Bernoulli trials. For $k \in \{0, 1, \ldots, n\}$ one has

$$Pr[Y = k] = \binom{n}{k} p^k (1 - p)^{n-k},$$

since there are $\binom{n}{k}$ combinations of n trials with k successes and $n - k$ failures. The probability of each combination is $p^k(1 - p)^{n-k}$. We have $E[X] = p$, $E[Y] = np$, $V[X] = p(1 - p)$ and $V[Y] = np(1 - p)$.

1.5. Random Numbers

Random numbers or random bits play an important role in cryptography. Keys are often chosen uniformly at random, and many cryptographic algorithms are probabilistic and require random input. One distinguishes between *true* random numbers in the sense of probability theory and *pseudorandom* numbers or bits. The latter are produced by deterministic algorithms, which take a short random input seed as input and generate a long output sequence that *appears* to be random. Pseudorandom generators are discussed in Section 2.8.

Definition 1.56. A random bit generator (RBG) is a mechanism or device which generates a sequence of random bits, such that the corresponding sequence of binary random variables $X_1, X_2, X_3, \dots$ has the following properties:

(1) $Pr[X_n = 0] = Pr[X_n = 1] = \frac{1}{2}$ for all $n \in \mathbb{N}$ (uniform distribution) and

(2) $X_1, X_2, \dots, X_n$ are mutually independent for all $n \in \mathbb{N}$.

Example 1.57. If at least one output bit is a combination of the other bits, for example if $X_3 = X_1 \oplus X_2$ (see Example 1.53), then this does not give a random bit sequence. This demonstrates that the obvious constructions to 'stretch' a given sequence cannot be used. ◊

Random bits or numbers can be produced manually (for example coin tossing, die rolling or mouse movements) or with *hardware random number generators*, which use physical phenomena like thermal noise, electrical noise or nuclear decay. Unfortunately, these mechanisms or devices tend to be slow, elaborate and/or costly. Fast all-digital random bit generators on current processor chips use thermal noise, but whether such generators can be trusted and do not have any weaknesses or even contain backdoors is disputed.

Remark 1.58. The required *uniform distribution* of the output of a bit generator can be achieved by *de-skewing* a possibly biased generator (see Example 1.59 below), but the statistical *independence* of the output bits is hard to achieve and difficult to prove.

Example 1.59. Von Neumann proposed the following de-skewing technique (*von Neumann extractor*): group the output bits into pairs, then turn 01 into 0 and 10 into 1. If the bits are independent, then the pairs 01 and 10 must have the same probability. The pairs 00 and 11 are discarded. The derived generator is slower but unbiased. ◊

The generation of *random bits* is basically equivalent to the generation of *random integer numbers*. In cryptography, random data is needed quite often. Random bits are used to seed deterministic pseudorandom generators (see Definition 2.32). In practice, the generators are often based on hash functions, HMAC or block ciphers in counter mode (see [**BK15**]). These cryptographic primitives are explained in the following chapters. The output of such generators appears totally unpredictable to an observer who does not know the seed, although deterministic functions are used to derive the output bits.

Random bits are needed to *generate keys*. Another application are *probabilistic* algorithms which take random data as auxiliary input. This may seem odd, but suppose an algorithm sometimes gives an incorrect result. Repeated application with fresh random input then reduces the failure rate. Furthermore, probabilistic algorithms ensure that the same input can have a different output value, and this can be a desirable property for data encryption.

Random numbers match surprisingly often. This is known as the *Birthday Problem* or *Birthday Paradox.*

Example 1.60. Assume that 23 people are in a certain place. Then the probability that at least two of them have their birthday on the same day of the year is *above* 50%. Intuitively, one would expect that around $\frac{365}{2}$ people would be needed for a probable birthday match. $\diamondsuit$

The explanation of this 'paradox' is quite simple: the probability p that no collision occurs (i.e., all birthdays are different) decreases exponentially with the number n of persons. We assume that birthdays are uniformly distributed. For $n = 2$, one has $p = \frac{364}{365}$. For $n = 3$, one gets $p = \frac{364}{365} \cdot \frac{363}{365}$, and each increment of n yields another factor. We write a SageMath function and obtain $p \approx 0.493$ for $n = 23$. The complementary probability $1-p$ for a birthday collision with 23 people therefore lies above 0.5.

```
sage: def birthday(k,n):
          p=1
          for i in range(1,k):
              p=p*(1-i/n)
          print RR(p)
sage: birthday(23,365)
0.492702765676015
```

Proposition 1.61. *Let Pr be a uniform distribution on a set Ω of cardinality n. If we draw $k = \left\lceil \sqrt{2\ln(2)n}\, \right\rceil \approx 1.2\sqrt{n}$ independent samples from Ω, then the probability of a collision is around* 50%.

Proof. See Exercise 16. $\square$

Example 1.62. Consider collisions of binary random strings of length l. The above Proposition 1.61 shows that collisions are expected after around $\sqrt{2^l} = 2^{l/2}$ values. If the strings are sufficiently long and uniformly distributed, then random collisions should almost never occur. For example, for 256-bit hash values with a uniform distribution, about 2^{128} hash values are required for a collision. On the other hand, only around 100,000 32-bit strings will probably have a collision. $\diamondsuit$

The running time of finding a collision among binary strings of length l is $O(2^{l/2})$. Unfortunately, a large amount of space is also required, since all $O(2^{l/2})$ strings have to be stored to detect a collision.

An optimization is possible if the samples are defined recursively by a function:

$$x_i = f(x_{i-1}) \text{ for } i \geq 1,$$

where x_0 is some initial value. Now the problem is to find a cycle in a sequence of iterated function values. *Floyd's cycle-finding algorithm* uses very little memory and is based on the following observation:

Proposition 1.63. *Let $f : X \to X$ be a function on some set X, $x_0 \in X$ and $x_i = f(x_{i-1})$ for $i \geq 1$. Suppose there exist $i, j \in \mathbb{N}$ such that $i < j$ and $x_i = x_j$. Then there exists an integer $k < j$ such that*

$$x_k = x_{2k}\,.$$

Proof. Let $\Delta = j - i$; then $x_i = x_{i+\Delta}$ and hence $x_k = x_{k+\Delta} = x_{k+m\Delta}$ for all integers $k \geq i$ and $m \geq 1$. Now let $k = m\Delta$, where $m\Delta$ is the smallest multiple of Δ that is also greater than or equal to i. The sequence $i, i+1, \dots, j-1$ of Δ consecutive integers contains the required number $k = m\Delta$. Therefore, $x_k = x_{2k}$ and $k = m\Delta < j$. □

Note that a collision must exist if X is a finite set. The above Proposition 1.63 implies that a collision in $x_0, x_1, \dots, x_j$ yields a collision of the special form $x_k = x_{2k}$ for some $k < j$. The least period of the sequence divides k. It is therefore sufficient to compute the pairs (x_i, x_{2i}) for $i = 1, 2, \dots$ until a collision occurs. These values can be recursively calculated:

$$x_i = f(x_{i-1}) \text{ and } x_{2i} = f(f(x_{2(i-1)})).$$

Assuming that the sequence $x_0, x_1, \dots$ is uniformly distributed and $|X| = n$, the running time is still $O(\sqrt{n})$, but now it is sufficient to store only two values. This approach is used in birthday attacks against hash functions and in *Pollard's ρ algorithms* for factoring and discrete logarithms. The sequence $x_0, x_1, \dots$ can be depicted by an initial tail and a cycle so that it looks like the greek letter ρ.

Remark 1.64. We only consider the part of Floyd's algorithm which finds a collision. The algorithm can also compute the least period, i.e., the length of the shortest cycle, and find the beginning of the cycle.

Example 1.65. Let $X = \mathbb{Z}_{107}$ be the set of residue classes modulo 107 and let

$$f(x) = x^2 + 26 \mod 107.$$

Set $x_0 \equiv 1 \mod 107$ and let $x_i = f(x_{i-1})$ for $i \geq 1$. We want to find a collision within the sequence $x_0, x_1, x_2, \dots$ and implement Floyd's cycle finding algorithm:

```
sage: def f(x):
            return(x*x+26)
sage: x=mod(1,107)
sage: y=mod(1,107)
sage: x=f(x)
sage: y=f(f(y))
sage: k=1
sage: while x!=y:
            x=f(x)
            y=f(f(y))
            k=k+1
            print  "k =",k," x =",x
k = 9  x = 39
```

Hence $x_9 = x_{18} = 39$ is a collision. Let's compute the first few elements of the sequence and verify the result:

```
sage: x=mod(1,107)
sage: for i in range(46):
            x=f(x)
            print("{:2}".format(x)),
27 6  62 18 29 11 40 21 39 49 73 5  51 59 83 67 21 39 49 73 5  51
59 83 67 21 39 49 73 5  51 59 83 67 21 39 49 73 5  51 59 83 67 21
```

The first seven elements form the initial segment (tail) of the sequence. The beginning of the cycle is $x_8 = 21$, and the sequence $21, 39, 49, \dots$ is cyclic of period 9.

1.6. Summary

- Sets, relations and functions are fundamental in mathematics and cryptography.
- Residue classes are defined by an equivalence relation on the integers. There are n different residue classes modulo n and the standard representatives are $0, 1, \dots, n-1$.
- Permutations are bijective functions on a finite set.
- The Big-O notation is used to give an asymptotic upper bound of the growth of a function in one variable.
- Algorithms with polynomial running time in terms of the input size are considered as efficient. The decision problems that can be solved in polynomial time form the complexity class **P**.
- Negligible functions are eventually smaller than any inverse polynomial.
- A discrete probability space consists of a countable sample space Ω and a probability distribution Pr on Ω. A real random variable X is a mapping from Ω to $\mathbb{R}$ and gives a probability distribution on $\mathbb{R}$.
- A true random bit generator can be described by a sequence of mutually independent binary and uniformly distributed random variables.
- The birthday paradox states that collisions of uniformly distributed random numbers occur surprisingly often.

Exercises

1. Let $X = ([-1, 1] \cap \mathbb{Z}) \times \{0, 1\}$. Enumerate the elements of X and determine $|X|$. Let $Y = \{1, 2, \ldots, |X|\}$. Give a bijection from X to Y.
2. Which of the following maps are *injective*, *surjective* or *bijective*? Determine the image $im(f)$ and give the inverse map f^{-1}, if possible.
   (a) $f_1 : \mathbb{N} \to \mathbb{N}$, $f_1(n) = 2n + 1$.
   (b) $f_2 : \mathbb{Z} \to \mathbb{N}$, $f_2(k) = |k| + 1$.
   (c) $f_3 : \{0, 1\}^8 \to \{0, 1\}^8$, $f_3(b) = b \oplus (01101011)$.
   (d) $f_4 : \{0, 1\}^8 \to \{0, 1\}^8$, $f_4(b) = b$ AND (01101011).
3. Show that the following sets are countable:
   (a) $\mathbb{Z}$.
   (b) $\mathbb{Z}^2$.
   (c) $\mathbb{Q}$.
   Hint: It is sufficient to construct an injective function into $\mathbb{N}$.
4. Let $f : X \to Y$ be a map between finite sets and suppose that $|X| = |Y|$. Show the following equivalences:
$$f \text{ is injective} \iff f \text{ is surjective} \iff f \text{ is bijective.}$$
5. Let $f : X \to Y$ be a function.
   (a) Let $B \subset Y$. Show that $f(f^{-1}(B)) \subset B$ with equality occurring if f is surjective.
   (b) Let $A \subset X$. Show that $A \subset f^{-1}(f(A))$ with equality occurring if f is injective.
6. Enumerate the integers modulo 26. Find the standard representative of the following integers in $\mathbb{Z}_{26}$:
$$-1000,\ -30,\ -1,\ 15,\ 2001,\ 293829329302932398231.$$
7. Consider the following relation S on $\mathbb{R}$:
$$S = \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x - y \in \mathbb{Z}\}.$$
   Show that S is an equivalence relation on $\mathbb{R}$. Determine the equivalence classes $\overline{0}$, $\overline{-2}$ and $\overline{\frac{4}{3}}$. Can you give an interval I such that there is a bijection between I and the quotient set $\mathbb{R}/\sim$?
8. Find an asymptotic upper bound of the following functions in n. Which of them are polynomial and which are negligible?
   (a) $f_1 = 2n^3 - 3n^2 + n$.
   (b) $f_2 = 3 \cdot 2^n - 2n + 1$.
   (c) $f_3 = \sqrt{2n + 1}$.
   (d) $f_4 = \frac{2n}{2^{n/2}}$.
   (e) $f_5 = \frac{5n^2 - n}{2n^2 + 3n + 1}$.

(f) $f_6 = 2^{\frac{1}{3}n+3}$.
(g) $f_7 = \log_2(n)^2 n$.

9. Let $f = f(x_0, x_1, x_2)$ be a 3-variable Boolean function with the following truth table:

x_2	x_1	x_0	$f(x)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Determine the algebraic normal form and the algebraic degree of f.

10. Suppose the number of operations of the most efficient attack against a cipher is

$$f(n) = e^{(2n^{1/3})},$$

where n is the key length. Why is $f(n)$ sub-exponential, but not polynomial in n? Compute *effective key length* $\log_2(f(n))$ for $n = 128$, $n = 1024$ and $n = 2048$.

11. Let $\Omega = \{0,1\}^8$ be a probability space with a uniform probability distribution. Compute the probability that a randomly chosen byte is balanced, i.e., contains four zeros and four ones.

12. Verify the expected values and variances in Example 1.55.

13. Two perfect dice are rolled and the random variables which give the numbers on the dice are called X and Y. Compute

$$Pr[X + Y \geq 10] \text{ and } Pr[X + Y \geq 10 \mid X - Y = 0].$$

Are the random variables $X + Y$ and $X - Y$ independent? Determine the expectation, the variance and the standard deviation of $X + Y$ and $X - Y$.

14. Suppose that 100 bits are generated uniformly at random and that additional bits are produced by XORing the preceding 100 bits. Are the new bits uniformly distributed? Does this construction give a random bit generator?

15. Let $X = \mathbb{Z}_{56807}$ and define a sequence by $x_0 \equiv 1$ and

$$x_i = x_{i-1}^2 + 1 \mod 56807$$

for $i \geq 1$. Use SageMath to find a collision and determine the least period of the sequence.

16. (Birthday Paradox) Let Pr be a uniform distribution on the sample space Ω with $|\Omega| = n$. If $k \leq n$ samples are independently chosen, then the probability p that all k values are different (i.e., no collision occurs) is

$$p = \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right).$$

(a) Show that the probability $1 - p$ of a collision satisfies

$$1 - p \geq 1 - e^{-\frac{k(k-1)}{2n}}.$$

(b) Determine the smallest number k such that $p \approx \frac{1}{2}$.

Hint: Use the inequality $1 - x \leq e^{-x}$ for $0 \leq x \leq 1$ and replace the factors $1 - \frac{i}{n}$ by $e^{-\frac{i}{n}}$. Compute the product and obtain a sum in the exponent. Use the formula $\sum_{i=1}^{k-1} i = \frac{k(k-1)}{2}$. For part (b), set $p = \frac{1}{2}$ and determine k using the quadratic formula. You may also approximate $k(k-1)$ by k^2. This gives the approximate number of samples needed for a probable collision.

Chapter 2

Encryption Schemes and Definitions of Security

This chapter contains the fundamental definitions of encryption schemes and their security. We look at security under different types of attacks and make assumptions about the computing power of adversaries. Then we study pseudorandom generators, functions and permutations, which are important primitives and the basis of many cryptographic constructions.

The definition of cryptosystems and some basic examples are given in Section 2.1. The following three sections deal with different types of security of encryption schemes. Perfect secrecy is the strongest type of security and is covered in Section 2.2. Since perfectly secure schemes can rarely be used in practice, we relax the requirements and consider computational security in Section 2.3. For the formal definition of security, we look at the success probability of polynomial-time adversaries in well-defined games or experiments in Section 2.4. We then explain the important definitions of eavesdropping (EAV) security, security against chosen plaintext attacks (CPA) and security against chosen ciphertext attacks (CCA). A secure scheme should have indistinguishable encryptions: challenged with a ciphertext and two possible plaintexts, a polynomial-time adversary fails to find the correct plaintext better than a random guesser.

We then turn to the construction of secure encryption schemes. Pseudorandom generators and families of pseudorandom functions and permutations are important building blocks of secure ciphers and are covered in Sections 2.8 and 2.9.

The combination of a family of functions or permutations and an operation mode defines an encryption scheme. In Section 2.10, we discuss ECB, CBC and CTR modes and their properties. The security of CBC or CTR mode encryption can be reduced to the pseudorandomness of the underlying block cipher.

The presentation of this chapter is heavily influenced by [**KL15**]. Other recommended references are [**BR05**], [**GB08**] and [**Gol01**].

2.1. Encryption Schemes

An *encryption scheme* essentially transforms plaintext into ciphertext and conversely. The algorithms are parametrized by keys and encryption is either *deterministic* or *randomized*, i.e., the ciphertext may depend on random input data. We use the symbols $:=$ or $=$ for a *deterministic* assignment, $\leftarrow$ for any type of assignment and $\overset{\$}{\leftarrow}$ for a *randomized* operation. For example, $c \overset{\$}{\leftarrow} \mathcal{E}_k(m)$ denotes an operation where an encryption algorithm $\mathcal{E}$ takes k, m and additional random data as input and outputs c. The symbol $\overset{\$}{\leftarrow}$ is also used for a uniform random choice from a set; for example $k \overset{\$}{\leftarrow} \{0,1\}^n$ gives a binary uniform random string k of length n.

Definition 2.1. An *encryption scheme* or *cryptosystem* consists of

- A plaintext space $\mathcal{M}$, the set of plaintext or clear-text messages.
- A ciphertext space $\mathcal{C}$, the set of ciphertext messages.
- A key space $\mathcal{K}$, the set of keys.
- A randomized key generation algorithm $Gen(1^n)$ that takes the security parameter n (see Remark 2.2 below) in unary form (compare Example 1.36) as input and returns a key $k \in \mathcal{K}$.
- An encryption algorithm $\mathcal{E} = \{\mathcal{E}_k \mid k \in \mathcal{K}\}$, which is possibly randomized. It takes a key and a plaintext message as input and returns the ciphertext or an error, if the plaintext is invalid. We write $c = \mathcal{E}_k(m)$ or $c \leftarrow \mathcal{E}_k(m)$, and $c \overset{\$}{\leftarrow} \mathcal{E}_k(m)$ if the algorithm is randomized. The error output is denoted by $\perp$.
- A deterministic decryption algorithm $\mathcal{D} = \{\mathcal{D}_k \mid k \in \mathcal{K}\}$ takes a key and a ciphertext message as input and returns the plaintext or an error symbol $\perp$ if the ciphertext is invalid. We write $m = \mathcal{D}_k(c)$ or $m \leftarrow \mathcal{D}_k(c)$.

We require that all algorithms are *polynomial* with respect to the input size. Since *Gen* takes a unary string $1^n = 1 \ldots 1$ of length n as input, the key generation algorithm is polynomial in n.

The scheme provides *correct decryption* if for each key $k \in \mathcal{K}$ and all plaintexts $m \in \mathcal{M}$ one has $\mathcal{D}_k(\mathcal{E}_k(m)) = m$ (see Figure 2.1). ♢

Remark 2.2. The *security parameter n* controls the security of the scheme and the difficulty to break it, as well as the run-time of key generation, encryption and decryption algorithms. The security parameter is closely related or even equal to the key length,

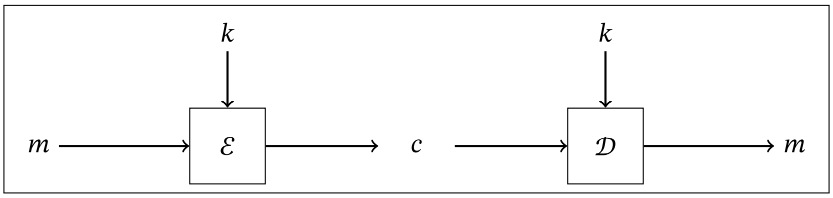


Figure 2.1. Encryption and decryption algorithms.

and quite often the key generation algorithm $Gen(1^n)$ outputs a uniform random key of length n. ◊

The scheme is said to be *symmetric-key* if encryption and decryption use the same secret key. In contrast, *public-key* (asymmetric-key) encryption schemes use *key pairs* $k = (pk, sk)$, where pk is public and sk is private; encryption takes the public key pk as input and decryption the private key sk (see Definition 9.1). The encryption algorithm $\mathcal{E}_{pk}$ must be carefully chosen so that the inversion (decryption) is computationally hard if only pk is known.

Until the 1970s, only symmetric-key schemes were known, but subsequently public-key methods became part of standard cryptography. We will see later that both schemes have their own field of application. Public-key encryption is studied in Chapter 9, while this chapter only deals with symmetric encryption schemes.

Remark 2.3. A cryptosystem should be secure under the assumption that an attacker knows the encryption and decryption algorithms. This is known as *Kerkhoff's Principle*. The security should be solely based on a secret key, not on the details of the system (see Exercise 2). ◊

In the past, the plaintexts, ciphertexts and keys were often constructed using the alphabet of letters. Now only the binary alphabet is relevant and

$$\mathcal{M},\ \mathcal{C},\ \mathcal{K} \text{ are subsets of } \{0,1\}^* = \bigcup_{n\in\mathbb{N}} \{0,1\}^n .$$

Modern symmetric encryption schemes support key lengths between 128 and 256 bits. In contrast, public-key algorithms (with the exception of elliptic curve schemes) use longer keys consisting of more than 1000 bits. Most modern symmetric schemes are able to encrypt plaintexts of arbitrary length. If however the message length is fixed by the security parameter, then we speak of a *fixed-length encryption scheme*.

Example 2.4. The *one-time pad* is an example of a simple but very powerful fixed-length symmetric encryption scheme. It uses the binary alphabet, and the key length is equal to the message length. The security parameter n defines the length of plaintexts, ciphertexts and keys:

$$\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0,1\}^n.$$

The key generation algorithm $Gen(1^n)$ outputs a uniform random key $k \overset{\$}{\leftarrow} \{0,1\}^n$. A key k of length n is *used only for one message*, $m \in \{0,1\}^n$. Encryption $\mathcal{E}_k$ and decryption $\mathcal{D}_k$ are identical and defined by a simple vectorial XOR operation:

$$c = \mathcal{E}_k(m) = m \oplus k, \qquad m = \mathcal{D}_k(c) = c \oplus k.$$

We will see below that this scheme provides perfect security, but since the key has the same length as the plaintext, the one-time pad is impractical. Much shorter keys (say several hundreds bits), which can be used for a large amount of data (say megabytes or gigabytes), are preferable.

Example 2.5. The *Vigenère cipher* of (key) length n is a classical example of a symmetric variable-length scheme over the alphabet of letters. One sets

$$\mathcal{M} = \mathcal{C} = \Sigma^* \text{ and } \mathcal{K} = \Sigma^n, \text{where } \Sigma = \{A, B, \dots, Z\} \cong \mathbb{Z}_{26}.$$

The letters A to Z can be represented by integers modulo 26 (see Example 1.21). The letter A corresponds to the residue class $\overline{0}$, B to $\overline{1}$, ..., Z to $\overline{25}$. A cyclic shift then becomes addition or subtraction of residue classes.

$Gen(1^n)$ generates a uniform random key string $k \overset{\$}{\leftarrow} \Sigma^n$ of length n. For encryption and decryption, the message and the ciphertext is split into blocks of length n, although the last block can be shorter. Each letter in a plaintext block is transformed by a cyclic shift, where the number of positions is determined by the corresponding key letter. For encryption the shifting is in the positive direction, and for decryption it is in the opposite direction.

$$c = \mathcal{E}_k(m) = \mathcal{E}_k(m_1 \| m_2 \| \dots) = (m_1 + k \parallel m_2 + k \parallel \dots) \bmod 26,$$
$$m = \mathcal{D}_k(c) = \mathcal{D}_k(c_1 \| c_2 \| \dots) = (c_1 - k \parallel c_2 - k \parallel \dots) \bmod 26.$$

For $n = 1$, one obtains a *monoalphabetic substitution cipher*, for example the so-called *Caesar cipher*, where $k = 3$: each letter is shifted by three positions, the letter A maps to D, B maps to E, etc. The Vigenère cipher of length $n > 1$ is an example of a *polyalphabetic substitution cipher*. Although the key can be long, each ciphertext letter only depends on a single plaintext character.

Example 2.6. A *Transposition Cipher* over an arbitrary alphabet encrypts a plaintext block of length n by reordering the characters. Keys are given by a random permutation of $\{1, 2, \dots, n\}$. *Bit permutations* are transposition ciphers over the binary alphabet. We observe that the frequency of characters is preserved by transposition ciphers. $\diamondsuit$

The above examples all use *linear or affine* transformations. Later we will see that affine ciphers are often vulnerable to known plaintext attacks which only require standard linear algebra (see Proposition 4.91). There are also classical *nonlinear* mono- or polyalphabetic substitution ciphers. If the plaintext contains text from a known language, then such ciphers can often be broken by a *frequency analysis*. It is also possible to reveal an unknown length of a polyalphabetic cipher.

Example 2.7. Another example of a polyalphabetic substitution cipher is the *Enigma rotor cipher machine* that was used since the 1920s and in particular by Nazi Germany during the Second World War. The Enigma is an electro-mechanical cipher machine that produces one ciphertext letter per keystroke. The key space is quite large and depends on initial rotor positions, ring settings and plug connections of the machine. The rotor positions change with each character. With considerable effort and cryptoanalytical machines, the Enigma was eventually broken. We recommend the book [**CM13**] for more details about the Enigma system.

2.2. Perfect Secrecy

One might want a cryptosystem that is *unbreakable* for any type of adversary, even if they have unlimited computing power. Such a scheme would provide *unconditional security*. Claude Shannon defined the notion of *perfect secrecy* [**Sha49**].

Definition 2.8. An encryption scheme is *perfectly secret* if for all plaintexts $m_0,\ m_1 \in \mathcal{M}$ and all ciphertexts $c \in \mathcal{C}$:

$$Pr[\mathcal{E}_k(m_0) = c\ ] = Pr[\mathcal{E}_k(m_1) = c\ ].$$

The probabilities are computed over randomly generated keys $k \in \mathcal{K}$. $\diamondsuit$

Perfect secrecy means that all plaintexts have the same probability for a given ciphertext. This property is also called *perfect indistinguishability*: without the secret key, it is impossible to find out which plaintext was encrypted. An eavesdropper truly learns nothing about the plaintext from the ciphertext, provided that any key is possible. In other words: given any ciphertext c, every plaintext message m is exactly as likely to be the underlying plaintext. Note that the definition requires a fixed plaintext length since encryption usually does not hide the length of a plaintext message.

The following lemma provides an alternative definition of perfect secrecy.

Lemma 2.9. *An encryption scheme is perfectly secret if and only if for every probability distribution over $\mathcal{M}$, every plaintext m and every ciphertext c for which $Pr[c] > 0$, the probability of m and the conditional probability of m given c coincide:*

$$Pr[m \mid c] = Pr[m]. \qquad \diamondsuit$$

Although perfect secrecy is a very strong requirement, there is a very simple cipher which achieves this level of security: the *one-time pad* (see Example 2.4).

Theorem 2.10. *The one-time pad is perfectly secret if the key is generated by a random bit generator* (*see Definition* 1.56) *and is only used once.*

Proof. Let n be a security parameter of the one-time pad. Suppose m_0, m_1 are plaintexts and c is a ciphertext of length n. Then there is exactly one key k_0 of length n which encrypts m_0 into c and in fact $k_0 = m_0 \oplus c$. Since we assumed an uniform distribution of keys, we have $Pr[\mathcal{E}_k(m_0) = c] = \frac{1}{2^n}$. The same holds true for m_1, which proves the Theorem. □

Example 2.11. Suppose a Vigenère cipher of key length 3 is used to encrypt four characters. Let $c = (y_1, y_2, y_3, y_4)$ be any ciphertext of length four, $m = (x_1, x_2, x_3, x_4)$ the corresponding plaintext and $\overline{k} = (k_1, k_2, k_3)$ the key. Then

$$y_1 \equiv x_1 + k_1,\ y_2 \equiv x_2 + k_2,\ y_3 \equiv x_3 + k_3,\ y_4 \equiv x_4 + k_1 \mod 26.$$

The difference between the fourth ciphertext and plaintext character is congruent to the difference between the first ciphertext and plaintext character:

$$k_1 \equiv y_1 - x_1 \equiv y_4 - x_4 \mod 26.$$

This forms a condition for all valid plaintext/ciphertext pairs. If $c = (y_1, y_2, y_3, y_4)$ is given, then there are many unfeasible plaintexts m_0, i.e., $\mathcal{E}_k(m_0) \neq c$ for all $k \in \mathcal{K}$, and therefore

$$Pr[\mathcal{E}_k(m_0) = c] = 0.$$

On the other hand, plaintexts $m_1 = (x_1, x_2, x_3, x_4)$ that satisfy the congruence $y_1 - x_1 \equiv y_4 - x_4 \mod 26$ are possible and their probability is

$$Pr[\mathcal{E}_k(m_1) = c] = \frac{1}{26^3}$$

if $k \in \mathbb{Z}_{26}^3$ is chosen uniformly at random. Hence the cipher does not have perfect secrecy. ◊

If the key is shorter than the message, then an encryption scheme cannot be perfectly secret: a known ciphertext message changes the posterior probability of a plaintext message. On the other hand, adversaries with limited resources might not be able to exploit this situation so that the scheme is still *computationally secure*. This is discussed in the next section.

2.3. Computational Security

Practical security should take the computing power of an adversary into account, rather than assuming unlimited resources. Furthermore, it is reasonable to accept a small chance of success, since an adversary might guess the correct plaintext or key. The computational approach estimates the probability of success in terms of time and computing resources.

We consider a scheme to be *broken* if an adversary can learn something about the plaintext from the ciphertext, i.e., if they obtain the output of any function of the plaintext, for example a plaintext bit or a sum of several plaintext bits. In the next section, breaking a scheme is defined in the context of an experiment, where an adversary has to answer a challenge.

Definition 2.12. A scheme is (t, ϵ)-secure if any adversary running in time t (measured in CPU cycles) can break the scheme with probability of ϵ at most.

Example 2.13. Assume that the best-known attack against a scheme is exhaustive key search (brute force) and that the key has length n. If testing a single key takes c CPU cycles and in total N CPU cycles are executed, then $\frac{N}{c}$ keys can be tested and the probability of success is approximately $\frac{N}{c2^n}$, if $\frac{N}{c} \ll 2^n$. Hence the scheme is $(N, \frac{N}{c2^n})$-secure. Suppose an adversary uses a computer with one 2 GHz CPU and performs a brute force attack against a scheme with 128-bit key length over the course of a year. Let's assume that $c = 1$. Then roughly 2^{55} keys can be tested and the scheme is $(2^{55}, 2^{-73})$-secure. Note that an event with a probability of 2^{-73} will never occur in practice. ♢

We see that concrete values depend on the hardware used, e.g., the type of CPU, as well as on the implementation of attacks. Now we give an *asymptotic* version of a definition of security. In this approach, the running time and the probability of breaking a scheme are considered as functions of the security parameter n (see Remark 2.2), and one analyzes the behavior for sufficiently large values of n.

Definition 2.14. An encryption scheme is called *computational secure* if every probabilistic algorithm with polynomial running time can only break the scheme with *negligible probability* in the security parameter n. ♢

Computational security only provides an *asymptotic security* guarantee, i.e., if the security parameter (see Remark 2.2) is sufficiently large.

Example 2.15. If the best possible attack is a brute-force search of a key of length n (see Example 2.13) and the running time is bounded by $N = p(n)$, where p is a polynomial, then the scheme is $(p(n), \frac{p(n)}{c \cdot 2^n})$-secure, where c is a constant. The probability $\frac{p(n)}{c \cdot 2^n}$ decreases exponentially to zero as n goes to infinity and is thus *negligible*. Therefore, the scheme is computationally secure.

2.4. Indistinguishable Encryptions

In Section 2.2 we saw that perfect secrecy requires *perfect indistinguishability*: for a given ciphertext, an adversary can only guess which plaintext was encrypted and all plaintexts (of a given length) occur with exactly the same probability. Now we want to relax the requirements of perfect secrecy for a more practical definition. Firstly, we consider only *efficient adversaries* running in polynomial time. Secondly, we allow a very small, i.e., negligible advantage over random guesses when an adversary tries to distinguish between two cases. *Indistinguishability* (IND) means that efficient adversaries are unable to find the correct plaintext out of two possibilities if the ciphertext is given. The performance of adversaries is not noticeably better than random guessing.

Furthermore, we want to define security under *different types of attacks*. We specify a *threat model* and consider attackers with certain capabilities: it is usually assumed that adversaries are able to *eavesdrop* on ciphertext messages, but we also look at adversaries who have access to plaintext/ciphertext pairs (*Known Plaintext Attack*) or can choose plaintexts (*Chosen Plaintext Attack*, CPA). If the adversary can even choose ciphertexts and obtain the corresponding plaintexts, then we call this a *Chosen Ciphertext Attack* (CCA).

We consider *experiments* (or *games*) between two algorithms, a polynomial-time adversary and a challenger. We denote the adversary by A and the challenger by C. The challenger takes as input a security parameter and sets up the experiment, for example by generating parameters and keys. C runs the experiment and interacts with A. In the experiment, A has certain choices and capabilities. Finally, A has to answer a challenge and outputs a single bit. The challenger verifies the answer and outputs 1 (A was successful and won the game) or 0 (A failed). Obviously, A has a 50% chance of randomly guessing the correct answer, but A might also use a more effective strategy. The game is repeated many times so that a success probability and an *advantage* (compared to random guesses) can be computed.

Such experiments may look artificial at first, but they answer the question as to whether an adversary can obtain at least one bit of secret information by applying an efficient algorithm. A scheme is considered broken if the probability of success is significantly higher than 50%.

In many security experiments, C chooses a uniform random secret bit b and A obtains a challenge that depends on b. Finally, A outputs a bit b' and wins the game if $b = b'$. Since the experiment is repeated many times, both b and b' can be considered as random variables. The following Table 2.1 contains the four combinations of b and b' and their joint probabilities:

Table 2.1. Joint probabilities of b and b'.

	$b' = 0$	$b' = 1$
$b = 0$	$Pr[b' = 0 \wedge b = 0]$	$Pr[b' = 1 \wedge b = 0]$
$b = 1$	$Pr[b' = 0 \wedge b = 1]$	$Pr[b' = 1 \wedge b = 1]$

If the adversary randomly guesses b', then all four probabilities are close to $\frac{1}{2}$. On the other hand, if A is doing a good job, then the diagonal entries are greater than $\frac{1}{2}$ and the other two are smaller than $\frac{1}{2}$.

We define A's *advantage* over random guesses as the difference between the *probability of success* (output of the experiment is 1) and the *probability of failure* (output of the experiment is 0):

$$\begin{aligned}\text{Adv}(A) &= |\, Pr[b' = 0 \wedge b = 0] + Pr[b' = 1 \wedge b = 1] \\ &\quad - Pr[b' = 1 \wedge b = 0] - Pr[b' = 0 \wedge b = 1]\,| \\ &= |\, Pr[b' = b] - Pr[b' \neq b]\,| \\ &= |\, Pr[Out(C) = 1] - Pr[Out(C) = 0]\,|\,.\end{aligned}$$

The difference could be negative, so we take the absolute value. A negative advantage would anyway imply a positive advantage for an inverse adversary A' who outputs 1 if A outputs 0 and vice versa.

The advantage is close to $\frac{1}{2} - \frac{1}{2} = 0$ if A randomly guesses b' and close to $1 - 0 = 1$ if A (or the inverse adversary) often succeeds in finding the correct answer.

Remark 2.16. The following definitions are also used in the literature:

$$\begin{aligned}\text{Adv}(A) &= 2 \cdot \left| Pr[b' = b] - \frac{1}{2} \right| = 2 \cdot \left| Pr[Out(C) = 1] - \frac{1}{2} \right|, \\ \text{Adv}(A) &= |\, Pr[b' = 1 | b = 1] - Pr[b' = 1 | b = 0]\,|.\end{aligned}$$

These formulas give the same advantage Adv(A) as above (see Exercise 6). The first formula is also used in the literature without the scaling factor 2.

2.5. Eavesdropping Attacks

Firstly, we assume a passive eavesdropper who only observes the ciphertext. This can be formalized by an experiment, where an adversary chooses two plaintexts and gets the ciphertext of one of them. Can the adversary find the correct plaintext and get an advantage over random guesses?

Definition 2.17. Suppose a symmetric encryption scheme is given and consider the following *indistinguishability experiment* (see Figure 2.2). A challenger takes the security parameter 1^n as input, generates a key $k \in \mathcal{K}$ by running $Gen(1^n)$ and chooses a random bit $b \overset{\$}{\leftarrow} \{0, 1\}$. A probabilistic polynomial-time adversary A is given 1^n, but neither k nor b are known to A. The adversary chooses two plaintexts m_0 and m_1 that are equal in length. The challenger returns the ciphertext $\mathcal{E}_k(m_b)$ of one of them. A tries to guess b (i.e., tries to find out which of the two plaintexts was encrypted) and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The *EAV advantage* of A is defined as

$$\text{Adv}^{\text{eav}}(A) = |\, Pr[b' = b] - Pr[b' \neq b]\,|\,.$$

The probability is taken over all random variables in this experiment, i.e., the key k, bit b, encryption $\mathcal{E}_k$ and randomness of A. ◊

A scheme is insecure under an EAV attack if the advantage of a smart adversary is not negligible, so that an adversary can successfully derive some information about the plaintext from the ciphertext.

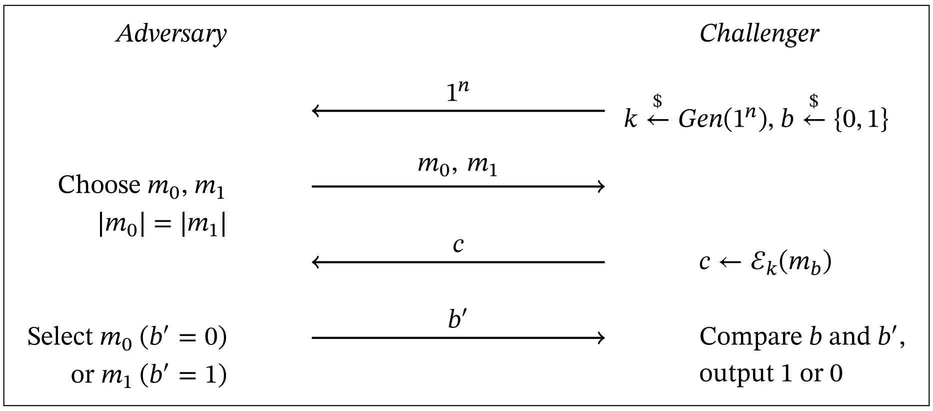


Figure 2.2. Indistinguishability experiment in the presence of an eavesdropper.

Definition 2.18. An encryption scheme has *indistinguishable encryptions in the presence of an eavesdropper* (IND-EAV secure or EAV-secure) if for every probabilistic polynomial-time adversary A, the advantage $\mathrm{Adv}^{\mathrm{eav}}(A)$ is negligible in the security parameter n. ♢

The definition can also be used to show that a particular scheme does *not have* EAV security. In this case, it suffices to give one example of a polynomial-time algorithm which achieves a non-negligible advantage in the EAV experiment.

Example 2.19. Suppose a scheme does not encrypt the first bit, i.e., the first plaintext bit and the first ciphertext bit coincide. Then the scheme does not have EAV security: an adversary could choose two plaintexts that differ in their first bit. In this way they are able to identify the correct plaintext from the challenge ciphertext.

Remark 2.20. It may seem surprising that the adversary can *choose the plaintexts* in an *eavesdropping attack*. It would be conceivable to let the challenger select the plaintexts. However, eavesdropping security should ensure that encryption protects *every plaintext*, not just selected or random plaintexts.

Furthermore, there are real-world situations where the plaintext space is rather small, say $\{0,\ 1\}$ or {YES, NO}. Such plaintexts also deserve protection. The EAV experiment is perfect for modeling this situation. ♢

We may also adopt a concrete approach to eavesdropping security (see Section 2.3):

Definition 2.21. An encryption scheme is (t, ϵ)-secure in the presence of an eavesdropper if for every probabilistic adversary A running in time t, the advantage of A is less than ϵ:

$$\mathrm{Adv}^{\mathrm{eav}}(A) < \epsilon. \qquad ♢$$

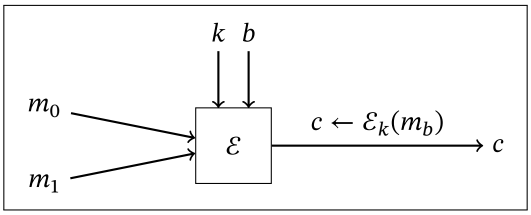


Figure 2.3. Indistinguishability: no efficient algorithm can tell the two cases apart if k and b are secret.

Remark 2.22. This is the first in a series of similar experiments and the reader is invited to think through the definitions. The adversary A chooses two plaintexts of the same length. The challenger encrypts one of the plaintexts and gives the ciphertext to the adversary. Then A has to *distinguish* between the two cases, i.e., find out which plaintext was encrypted (see Figure 2.3). The question is whether A finds a clever way to tackle the challenge, or else falls back on random guesses. The advantage is finally computed over many games with sample keys k, random bits b, ciphertexts c and other randomness used by the adversary.

In practice, one would not really conduct a large number of such experiments. In particular, modeling the possible strategies of an adversary seems rather difficult, but the above definition is useful since it clearly states the *requirements for eavesdropping security* and defines under which condition a *scheme is considered broken*.

Remark 2.23. Our security definition is based on *indistinguishability*. One can show that this is equivalent to *semantic security*: an adversary cannot learn any partial information about the plaintext from the ciphertext. This means that any function of the plaintext, say extracting one bit, is hard to compute and polynomial-time adversaries cannot do any better than random guessing. We refer to the literature for more details on semantic security ([**KL15**], [**Gol01**]).

2.6. Chosen Plaintext Attacks

Now we turn to a more powerful adversary who has flexible access to an encryption oracle: they can freely choose plaintexts and obtain the corresponding ciphertexts. This might help to decrypt a challenge ciphertext, at least if there are only two plaintext candidates. Can they do better than randomly guessing the correct plaintext? This is measured by the IND-CPA advantage, which is defined analogously to the EAV-advantage above.

Definition 2.24. Suppose a symmetric encryption scheme is given and an adversary A has access to an encryption oracle. Consider the following experiment (see Figure 2.4). A challenger takes the security parameter 1^n as input, generates a key $k \in \mathcal{K}$ and chooses a uniform random bit $b \overset{\$}{\leftarrow} \{0,1\}$. A probabilistic polynomial-time adversary A is given 1^n, but k and b are not known to A. The adversary can choose arbitrary plaintexts and get the corresponding ciphertext from an encryption oracle. The adversary then chooses two different plaintexts m_0 and m_1 of the same length. The challenger returns the ciphertext $\mathcal{E}_k(m_b)$ of one of them. The adversary A continues to have access to the encryption oracle. Finally, A tries to guess b and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The *IND-CPA advantage* of A is defined as

$$\mathrm{Adv}^{\mathrm{ind-cpa}}(A) = |\, Pr[b' = b] - Pr[b' \neq b]\,| \,.$$

The probability is taken over all random variables in this experiment, i.e., the key k, bit b, encryption $\mathcal{E}_k$ and randomness of A.

Definition 2.25. A scheme has *indistinguishable encryptions under chosen plaintext attack* (IND-CPA secure or CPA-secure) if for every probabilistic polynomial-time adversary A, the advantage $\mathrm{Adv}^{\mathrm{ind-cpa}}(A)$ is negligible in n. ◊

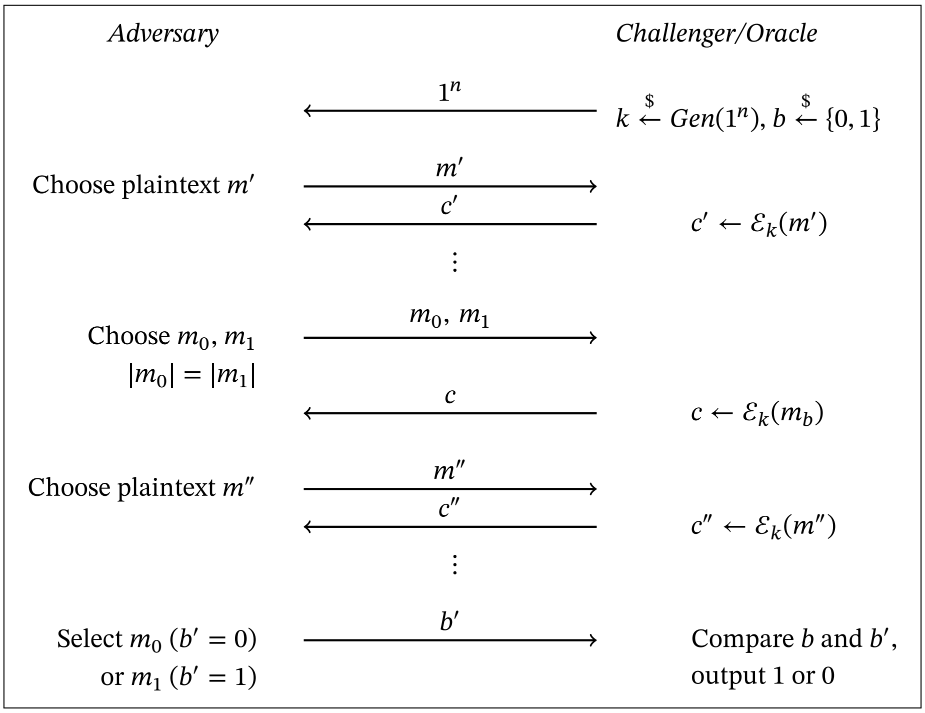

Adversary
Challenger/Oracle
1^n
$k \overset{\$}{\leftarrow} Gen(1^n),\ b \overset{\$}{\leftarrow} \{0,1\}$
Choose plaintext m'
m'
c'
$c' \leftarrow \mathcal{E}_k(m')$
Choose m_0, m_1
$|m_0| = |m_1|$
m_0, m_1
c
$c \leftarrow \mathcal{E}_k(m_b)$
Choose plaintext m''
m''
c''
$c'' \leftarrow \mathcal{E}_k(m'')$
Select m_0 ($b' = 0$) or m_1 ($b' = 1$)
b'
Compare b and b', output 1 or 0

Figure 2.4. CPA indistinguishability experiment. The adversary may repeatedly ask for the encryption of chosen plaintexts m', m''.

Security under chosen plaintext attack is quite a strong condition. It basically says that an adversary is not able to obtain a single bit of information from a given ciphertext, even if they can ask for the encryption of arbitrary plaintexts.

Remark 2.26. The definition immediately implies that a *deterministic encryption scheme cannot be secure under IND-CPA*. An adversary can, in fact, ask the oracle for the encryption of the chosen plaintexts m_0 and m_1. Then they only need to compare the returned ciphertext with the challenge ciphertext. If the scheme is deterministic, the IND-CPA advantage is equal to 1, but for a non-deterministic scheme, each encryption of a fixed plaintext can yield another ciphertext. Therefore, a simple comparison cannot be used in the non-deterministic case.

Example 2.27. Suppose an encryption scheme is probabilistic, but the *first ciphertext bit* depends only on the plaintext and the key and not on the randomness of the encryption algorithm. If the first ciphertext bit is not constant, then the scheme is *not secure under IND-CPA*, even if the method is very strong otherwise. An adversary would generate multiple plaintexts of the same length and ask for encryption, until they find two plaintexts m_0, m_1 such that the corresponding ciphertexts differ in their first bit. Then they choose m_0 and m_1 in the CPA experiment. The first bit of the challenge ciphertext reveals which of the two plaintexts was encrypted by the challenger.

Remark 2.28. Constant patterns in the ciphertext do not violate IND-CPA (or IND-EAV) security, and it is not required that the ciphertext looks like a random sequence. In fact, an adversary cannot leverage constant parts of the ciphertext in order to obtain information about the plaintext. $\diamondsuit$

In practice, one wants to encrypt *multiple messages* with the same key. This is not directly addressed in the EAV and CPA experiments, where an adversary has to find the plaintext for a *single* ciphertext message. The generalization of these games for multiple encryptions allows the adversary to provide *multiple pairs of plaintext messages*. A *left-or-right oracle* encrypts either the left or the right plaintext of each pair (depending on the secret bit b) and returns the ciphertexts.

In fact, EAV security for multiple messages is stronger than EAV security for a single message (see Exercise 11). On the other hand, CPA-secure schemes remain secure when the adversary has access to a left-or-right encryption oracle:

Proposition 2.29. *If an encryption scheme is CPA-secure* (*see Definition* 2.25)*, then it is CPA-secure for multiple encryptions.*

2.7. Chosen Ciphertext Attacks

After discussing EAV and CPA security, we now consider even more powerful attackers. Security under a *chosen ciphertext attack* is defined in a similar way to IND-CPA security, but additionally gives the adversary the power to request the decryption of any *chosen ciphertext* except the challenge ciphertext.

Definition 2.30. Suppose a symmetric encryption scheme is given. Consider the following experiment (see Figure 2.5). On input 1^n a challenger generates a random key $k \in \mathcal{K}$ and a random bit $b \overset{\$}{\leftarrow} \{0,1\}$. A probabilistic polynomial-time adversary A is given 1^n, but k and b are not known to A. The adversary can ask an oracle to encrypt arbitrary plaintexts and to decrypt ciphertexts. The adversary chooses two different plaintexts m_0 and m_1 of the same length. The challenger returns the ciphertext $c = \mathcal{E}_k(m_b)$ of one of them. The adversary A continues to have access to the encryption and decryption oracle, only decryption of the challenge ciphertext c is not permitted. Finally, A tries to guess b and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. Then the *IND-CCA advantage* of A is defined by

$$\text{Adv}^{\text{ind-cca}}(A) = |\, Pr[b' = b] - Pr[b' \neq b]\,| \,.$$

The probability is taken over all random variables in this experiment, i.e., the key k, bit b, encryption $\mathcal{E}_k$ and randomness of A. $\lozenge$

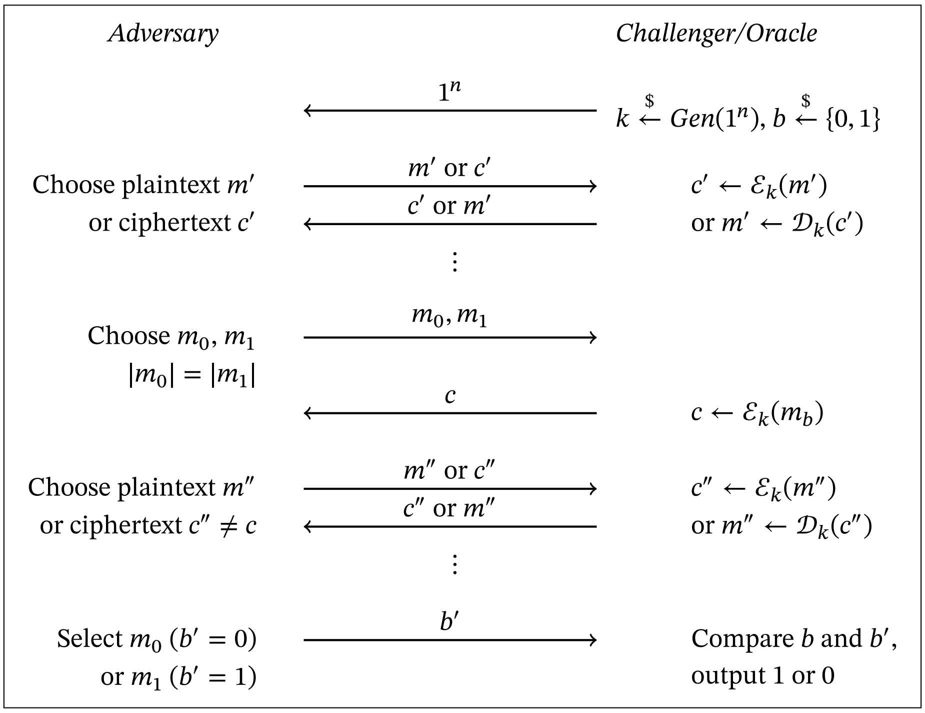

Adversary
Challenger/Oracle
1^n
$k \overset{\$}{\leftarrow} Gen(1^n),\, b \overset{\$}{\leftarrow} \{0,1\}$
Choose plaintext m' or ciphertext c'
m' or c'
c' or m'
$c' \leftarrow \mathcal{E}_k(m')$ or $m' \leftarrow \mathcal{D}_k(c')$
Choose m_0, m_1 $|m_0| = |m_1|$
m_0, m_1
c
$c \leftarrow \mathcal{E}_k(m_b)$
Choose plaintext m'' or ciphertext $c'' \neq c$
m'' or c''
c'' or m''
$c'' \leftarrow \mathcal{E}_k(m'')$ or $m'' \leftarrow \mathcal{D}_k(c'')$
Select m_0 $(b' = 0)$ or m_1 $(b' = 1)$
b'
Compare b and b', output 1 or 0

Figure 2.5. CCA2 indistinguishability experiment. The adversary may repeatedly ask for the encryption of chosen plaintexts m', m'' and for the decryption of chosen ciphertexts c', c'' except the challenge c.

Definition 2.31. A scheme has *indistinguishable encryptions under adaptive chosen ciphertext attack* (IND-CCA2 secure or CCA2-secure) if for every probabilistic polynomial-time adversary A, the advantage $\mathrm{Adv}^{\mathrm{ind-cca}}(A)$ is negligible in n. ◊

In the literature, one distinguishes between IND-CCA1 and IND-CCA2. In the IND-CCA1 experiment, the attack is not adaptive and the adversary may use the decryption oracle only *before* being given the challenge. In contrast, the above IND-CCA2 experiment allows the adversary to adapt their queries to the challenge. CCA2 security is stronger than CCA1 security, and CCA security mostly refers to the CCA2 experiment.

CCA2-security requires *non-malleability*. We say a ciphertext c is *malleable* if an adversary can produce a valid ciphertext $c' \neq c$ such that the corresponding plaintexts m and m' are related. Note that tampering with the ciphertext does not require decryption. Standard examples of malleability are encryption schemes where flipping one ciphertext bit changes only one plaintext bit, e.g., a block cipher in CTR mode (see Definition 2.48 below).

2.8. Pseudorandom Generators

Randomness and pseudorandomness play a fundamental role in cryptography, especially for encryption, but also for other cryptographic operations. A pseudorandom string *looks like* a uniform string without being the output of a random bit generator (compare Section 1.5). A *pseudorandom generator* is a deterministic algorithm that stretches a short truly random seed and outputs a long output sequence that *appears to be uniform random*. Pseudorandom generators are basic building blocks for many cryptographic constructions.

Definition 2.32. Let G be a deterministic polynomial-time algorithm that takes an input seed $s \in \{0,1\}^n$ and outputs a string $G(s) \in \{0,1\}^{l(n)}$, where $l(.)$ is a polynomial and $l(n) > n$ for all $n \in \mathbb{N}$. Then G is called a *pseudorandom generator* (prg) if the output cannot be distinguished from a uniform random sequence in polynomial time. Consider the following experiment (see Figure 2.6): on input 1^n a seed $s \overset{\$}{\leftarrow} \{0,1\}^n$ and a bit $b \overset{\$}{\leftarrow} \{0,1\}$ are chosen uniformly at random. A probabilistic polynomial-time adversary A obtains 1^n, but knows neither s nor b. A challenger chooses a uniform random string $r \overset{\$}{\leftarrow} \{0,1\}^{l(n)}$. If $b = 0$ then $c = r$ is given to A. Otherwise, A receives $c = G(s)$. The adversary tries to distinguish between the two cases and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. Then the prg-advantage of A is defined as

$$\mathrm{Adv}^{\mathrm{prg}}(A) = |\, Pr[b' = b] - Pr[b' \neq b]\,| \,.$$

The probability is taken over all random variables in this experiment, i.e., the seed s, bit b, string r and randomness of A.

G is called a *pseudorandom generator* if for all probabilistic polynomial-time distinguisher A, the prg-advantage is negligible in n.

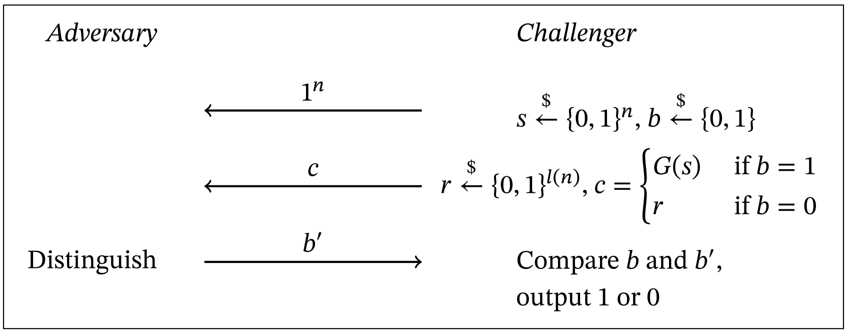

Adversary
Challenger
1^n
$s \overset{\$}{\leftarrow} \{0,1\}^n,\ b \overset{\$}{\leftarrow} \{0,1\}$
c
$r \overset{\$}{\leftarrow} \{0,1\}^{l(n)},\ c = \begin{cases} G(s) & \text{if } b = 1 \\ r & \text{if } b = 0 \end{cases}$
Distinguish
b'
Compare b and b', output 1 or 0

Figure 2.6. Distinguishability experiment for a pseudorandom generator G.

Remark 2.33. A pseudorandom generator is never a truly random bit generator! Because of the expansion from length n to $l(n) > n$, the distribution of output values cannot be uniform. In fact, many strings of length $l(n)$ do not occur in the image of G since the domain $\{0,1\}^n$ is too small, but with limited resources, the output of G *looks random* and cannot be distinguished from a truly random sequence (see Figure 2.7).$\diamondsuit$

The construction of a pseudorandom generator is a non-trivial task. Obvious constructions, like XORing some seed bits in order to produce new bits, do not work since the generated sequence can easily be distinguished from a truly random sequence. The output must be *unpredictable*, i.e., a polynomial-time adversary who is given the first i bits of the output, is unable to predict the $(i+1)^{\text{st}}$ bit with a probability noticeably higher than 50%. In other words, the generator must pass the *next-bit test*. One can show the following Theorem (see [**Gol01**], Theorem 3.3.7):

Theorem 2.34. *A generator is pseudorandom if and only if it is unpredictable in polynomial time.* $\diamondsuit$

Proving that a pseudorandom generator is unpredictable is straightforward (see Exercise 12), but the opposite direction is harder.

Remark 2.35. An unpredictable generator is also called a *cryptographically secure pseudorandom number generator* (CSPRNG or CPRNG). There is a statistical test suite for pseudorandom generators produced by the American NIST [**RSN⁺10**], but note that passing the complete test battery does not prove the pseudorandomness. On the other hand, failing one of the tests indicates that the generator is not pseudorandom (unless the deviation from the expected result is in turn random). $\diamondsuit$

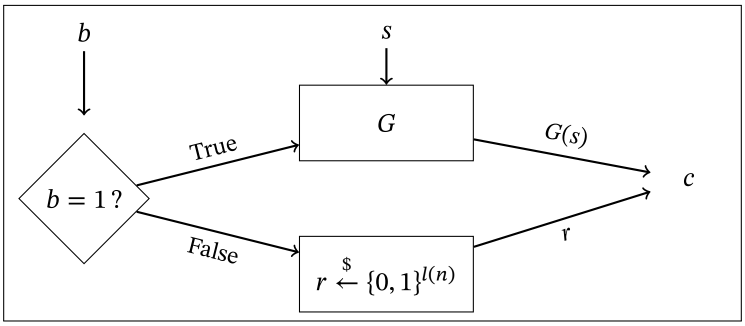


Figure 2.7. Pseudorandomness: adversaries cannot tell the two cases apart if b and s are secret.

The output of a pseudorandom generator can be used as a *keystream*. Like the one-time pad, XORing the plaintext and the keystream defines a fixed-length encryption scheme. This type of scheme is called a *stream cipher*. Further details on stream ciphers can be found in Chapter 6.

Definition 2.36. Let $l(.)$ be a polynomial. A pseudorandom generator G, which on input $k \in \{0,1\}^n$ produces an output sequence $G(k) \in \{0,1\}^{l(n)}$, defines a fixed-length encryption scheme by the following construction:

The key generation algorithm $Gen(1^n)$ takes the security parameter 1^n as input and outputs a uniform random key $k \overset{\$}{\leftarrow} \{0,1\}^n$ of length n. Set $\mathcal{M} = \mathcal{C} = \{0,1\}^{l(n)}$. Encryption $\mathcal{E}_k$ and decryption $\mathcal{D}_k$ are identical and are defined by XORing the output stream $G(s)$ with the input data:

$$c = \mathcal{E}_k(m) = m \oplus G(k) \quad \text{and} \quad m = \mathcal{D}_k(c) = c \oplus G(k). \qquad \diamondsuit$$

Theorem 2.37. *If G is a pseudorandom generator, then the associated encryption scheme has indistinguishable encryptions in the presence of an eavesdropper (EAV-secure).*

Proof. We only sketch a *proof by reduction*. For more details we refer the reader to [**KL15**].

Suppose the encryption scheme is not EAV-secure; then there is a polynomial-time algorithm A with a non-negligible advantage $\mathrm{Adv}^{\mathrm{eav}}(A)$ in the EAV experiment (see Definition 2.18). We construct an adversary B in the prg distinguishability experiment which uses A as a subroutine. B obtains a challenge string c of length $l(n)$ and has to determine whether c was generated by G or chosen uniformly. Now B runs A, chooses a uniform bit $b \overset{\$}{\leftarrow} \{0,1\}$ and obtains a pair m_0, m_1 of messages of length $l(n)$ from A. Subsequently, B gives the challenge $c \oplus m_b$ to A. Finally, A outputs b' and B observes whether or not A succeeds. Remember that we assumed that A does a good job in the

EAV experiment, and so a correct output of A indicates that c was produced by the generator G. Therefore, B outputs 1, i.e., B guesses that $c = G(s)$ if A succeeds ($b = b'$). Otherwise, B outputs 0, i.e., B guesses that c is a random string.

If $\mathrm{Adv}^{\mathrm{eav}}(A)$ is non-negligible, then $\mathrm{Adv}^{\mathrm{prg}}(B)$ is non-negligible, too. Furthermore, B runs in polynomial time. This contradicts our assumption that G is a pseudorandom generator and proves the theorem. □

Now we know that the construction described in Definition 2.36 has a security guarantee, but a disadvantage is that the message length $l(n)$ is fixed for a given security parameter n. Furthermore, the above encryption scheme is not CPA-secure (since it is deterministic) and not EAV-secure for *multiple encryptions* with the same seed or key (see Exercise 11). Like the one-time pad, the same keystream *must not be re-used* for two or more plaintexts.

In Chapter 6, we will deal with *variable-length* pseudorandom generators and *stream ciphers*. They take a seed (or key) and an initialization vector as input, use an internal state and recursively generate as many output bits as required.

2.9. Pseudorandom Functions

CPA-secure schemes are often based on *pseudorandom functions and permutations*. Below, we look at the definition of such a family of functions. Constructing pseudorandom functions and permutations is a difficult task and we will have to wait until Section 5.2 for a concrete instance (AES).

We consider a family of functions:

$$F = F(n) : \mathcal{K}_n \times D_n \to R_n,$$

where n is a security parameter, $\mathcal{K}_n = \{0,1\}^n$ the set of keys, $D_n = \{0,1\}^{l_1(n)}$ the domain, $R_n = \{0,1\}^{l_2(n)}$ the range and $l_1(.)$, $l_2(.)$ are polynomials in n. We suppose that F can be computed in *polynomial time*. Fixing $k \in \mathcal{K}_n$ yields a function $F_k : D_n \to R_n$.

The family F is called *pseudorandom* if the functions $F_k : D_n \to R_n$ appear to be random for a polynomial-time adversary who knows the input-output behavior of F_k, but is not given the secret key k. In other words, a polynomial-time adversary is not able to distinguish a pseudorandom function from a truly random function.

To simplify the notation we assume that the key length is n (the security parameter) and that the input and output lengths are equal. We write $l_1(n) = l_2(n) = l$ and bear in mind that l depends on n. Then $D_n = R_n = \{0,1\}^l$ and

$$F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l.$$

Definition 2.38. Let F be a public family of functions as outlined above. Consider the following experiment (see Figure 2.8): on input 1^n a random key $k \overset{\$}{\leftarrow} \mathcal{K}$ of length n and a random bit $b \overset{\$}{\leftarrow} \{0,1\}$ are chosen. A polynomial-time adversary A obtains 1^n, but knows neither k nor b. Furthermore, a challenger chooses a uniform random function $f_0 : \{0,1\}^l \to \{0,1\}^l$ and sets $f = f_0$ if $b = 0$ and $f = F_k$ if $b = 1$. The adversary has oracle access to f, i.e., A can choose input values and obtain the output of f. The adversary tries to find out which function was used and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The *prf-advantage* of A is defined as

$$\mathrm{Adv}^{\mathrm{prf}}(A) = |\,Pr[b' = b] - Pr[b' \neq b]\,|.$$

The probability is taken over all random variables in this experiment, i.e., the key k, bit b, function f_0 and randomness of A. ◊

If the adversary is not able to distinguish between these functions, then the advantage is close to 0. Pseudorandom functions have the property that the prf-advantage is negligible in n.

Definition 2.39. A keyed function family F as described above is called a *pseudorandom function* (prf) if, for every probabilistic polynomial time adversary A, the prf-advantage $\mathrm{Adv}^{\mathrm{prf}}(A)$ is negligible in n. ◊

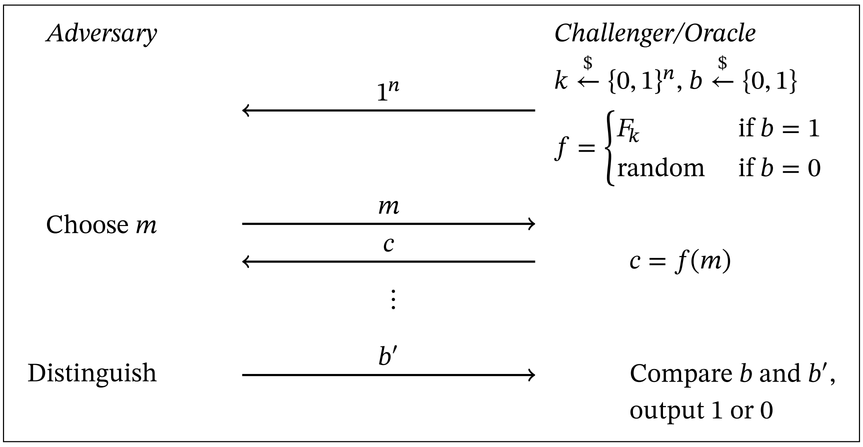

Adversary
Challenger/Oracle
$k \xleftarrow{\$} \{0,1\}^n,\ b \xleftarrow{\$} \{0,1\}$
1^n
$f = \begin{cases} F_k & \text{if } b = 1 \\ \text{random} & \text{if } b = 0 \end{cases}$
Choose m
m
c
$c = f(m)$
$\vdots$
Distinguish
b'
Compare b and b', output 1 or 0

Figure 2.8. Distinguishability experiment for a pseudorandom function. The adversary can ask for multiple values of f.

Pseudorandomness is a strong property that cannot be achieved by simple constructions. In Chapter 4 we will see that linear or affine families of functions cannot be pseudorandom (Proposition 4.93).

Remark 2.40. A pseudorandom generator G can be derived from a pseudorandom function F by the following construction: choose a uniform random counter ctr and set

$$G(k) = F_k(ctr + 1) \,||\, F_k(ctr + 2) \,||\, F_k(ctr + 3) \,||\, \dots,$$

where *ctr* is viewed as an integer and addition is done modulo 2^n. Incrementing the counter allows you to generate output of (almost) arbitrary length. The opposite, i.e., constructing a pseudorandom function from a pseudorandom generator, is also possible but more elaborate. In practice, one prefers pseudorandom *functions* as a basic building block. ◊

Pseudorandom permutations are an important special case of pseudorandom functions. They are given by a keyed family of *permutations*:

$$F = F(n) : \mathcal{K}_n \times D_n \to D_n.$$

For any $k \in \mathcal{K}$, the function $F_k : D_n \to D_n$ is a permutation, i.e., F_k is bijective. As above, we assume that the key length is n and the strings in D are of length $l(n) = l$, i.e. $\mathcal{K}_n = \{0,1\}^n$, $D_n = \{0,1\}^l$ and

$$F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l.$$

A family F of permutations is called *pseudorandom* if polynomial-time adversaries are not able to distinguish F from a truly random permutation. The following definition is analogous to Definition 2.38.

Definition 2.41. Let F be a public family of permutations. Consider the following experiment: on input 1^n a random key $k \overset{\$}{\leftarrow} \mathcal{K}$ of length n and a random bit $b \overset{\$}{\leftarrow} \{0,1\}$ are generated. A polynomial-time adversary A obtains 1^n, but knows neither k nor b. Furthermore, a challenger chooses a uniform random permutation f_0 of $\{0,1\}^n$ and sets $f = f_0$ if $b = 0$ and $f = F_k$ if $b = 1$. The adversary has oracle access to f, i.e., A can choose input values and obtain the output of f. The adversary tries to find out which function was used and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The *prp-cpa-advantage* of A is defined as

$$\mathrm{Adv}^{\mathrm{prp-cpa}}(A) = |\, Pr[b' = b] - Pr[b' \neq b] \,| \,.$$

The probability is taken over all random variables in this experiment, i.e., the key k, bit b, permutation f_0 and randomness of A. ◊

We may also give the adversary additional access to the inverse permutation.

Definition 2.42. A family of permutations F, as described above, is called a *pseudorandom permutation* (prp) if, for every probabilistic polynomial time adversary A, the prp-cpa-advantage $\mathrm{Adv}^{\mathrm{prp-cpa}}(A)$ is negligible in n. If adversaries also have oracle access to the inverse function f^{-1} (see Figure 2.9) and the advantage is negligible, then F is said to be a *strong pseudorandom permutation*. ◊

Remark 2.43. A pseudorandom permutation is also a pseudorandom function. The *prp* and *prf* advantages are almost identical if domain and range coincide, but there is one issue: suppose we want to use a pseudorandom *permutation* as a pseudorandom *function*. Since permutations do not have any collisions (different inputs must

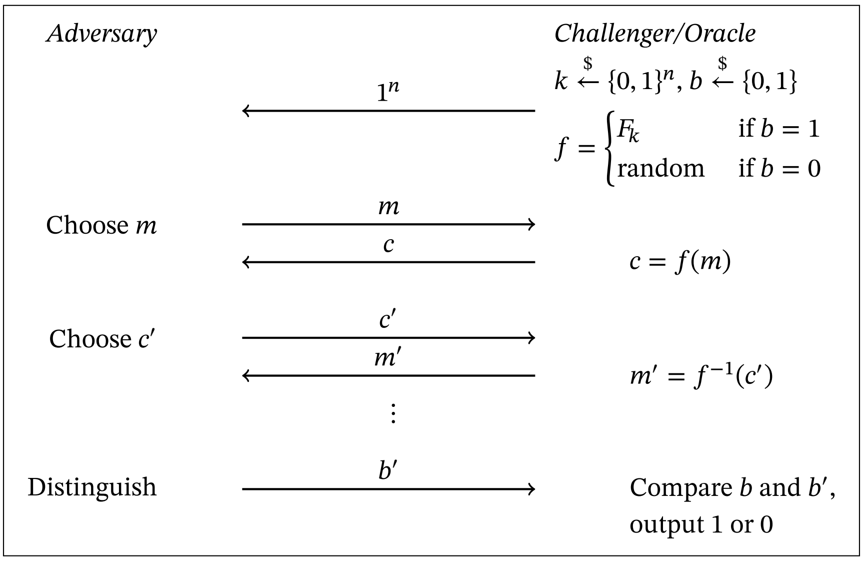

Adversary
Challenger/Oracle
$k \xleftarrow{\$} \{0,1\}^n,\ b \xleftarrow{\$} \{0,1\}$
1^n
$f = \begin{cases} F_k & \text{if } b = 1 \\ \text{random} & \text{if } b = 0 \end{cases}$
Choose m
m
c
$c = f(m)$
Choose c'
c'
m'
$m' = f^{-1}(c')$
$\vdots$
Distinguish
b'
Compare b and b', output 1 or 0

Figure 2.9. Distinguishability experiment for a strong pseudorandom permutation. The adversary can ask for multiple values of f and f^{-1}.

have different outputs), an adversary might distinguish the permutation from a random function by computing many input-output pairs and searching for collisions. Suppose that the domain is $D = \{0,1\}^l$. By the *Birthday Paradox* (see Proposition 1.61), a random *function* will probably have collisions after around $2^{l/2}$ samples. However, a polynomial-time adversary is not able to check an exponential number of values. In fact, the *prp/prf switching lemma* [**BR06**] states that the difference between the *prp* and *prf* advantages is negligible.

Remark 2.44. In the experiments for pseudorandom functions and permutations, the secret key k is fixed and an adversary can only obtain input-output examples for that key. Now, under a *related-key attack* (RKA) they can also study the input-output behavior for keys k_1, k_2, ... *related* to k. In an RKA experiment one has to specify which keys are related. Examples include keys with a fixed difference Δ to k, i.e. $k + \Delta$, or the complemented key $\overline{k}$, where all key bits are flipped. Note that k and its related keys are still unknown to an adversary, but now they can ask the oracle to use a related key instead of k. This gives the adversary more power and *pseudorandomness with respect to a related-key attack* is stronger than the usual notion of a prp or prf. We refer the reader to [**BK03**] for details.

Example 2.45. Suppose a function family has the following complementation property for all keys k and input m:

$$\overline{F_k(m)} = F_{\overline{k}}(\overline{m}).$$

If the oracle accepts the complemented key $\overline{k}$ as related to k, then an adversary can easily distinguish F from a random function simply by checking the above equation.

It is known that the former Data Encryption Standard (DES) has the complementation property and the cipher is thus vulnerable to related-key attacks. ◊

In the next section, we explain how a pseudorandom permutation can be turned into an encryption scheme. In practice, one selects a *block cipher*, for example AES (see Section 5.2), and combines it with an *operation mode* in order to obtain an encryption scheme. The block cipher is modeled to behave as a pseudorandom permutation. It should be noted however that there is no hard proof or guarantee (based on well-known assumptions) that such a block cipher really is a pseudorandom permutation. Number-theoretic *one-way permutations* can be used to define pseudorandom permutations, but they are less efficient and are only used in public-key cryptography (see Chapter 9).

2.10. Block Ciphers and Operation Modes

Block ciphers are a very important building block of modern encryption schemes.

Definition 2.46. A family of permutations

$$E = E(n) \,:\, \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$$

is said to be a *block cipher*. n is the security parameter and the *key length*, and $\{0,1\}^n$ is the key space. $l = l(n)$ is called the *block length*. ◊

Block ciphers can be thought of as concrete instances of families of *pseudorandom permutations*. Each key $k \in \{0,1\}^n$ yields a bijective function

$$E_k \,:\, \{0,1\}^l \to \{0,1\}^l,$$

i.e., a permutation of $\{0,1\}^l$. Key and block lengths are often fixed or restricted to a few values, for example $l = 128$ and $n \in \{128, 192, 256\}$. Nevertheless, secure block ciphers should behave as pseudorandom permutations. A specific example of an important block cipher (AES) is studied in Section 5.2.

Block ciphers only encrypt messages of a fixed block length l, but the combination of a block cipher and an *operation mode* defines a variable-length encryption scheme. Below, we present ECB, CBC and Counter (CTR) modes (see [**Dwo01**]). Additional modes are dealt with in subsequent chapters: OFB and CFB modes in Section 6.1 and the GCM mode in Section 8.4.

Definition 2.47. (ECB and CBC mode) Let

$$E \,:\, \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$$

be a block cipher. We define two variable-length symmetric encryption schemes based on E. In both cases, the key generation algorithm $Gen(1^n)$ outputs a uniform random key $k \overset{\$}{\leftarrow} \{0,1\}^n$ of length n. A binary plaintext message m is split into blocks of length

$l = l(n)$ and a padding mechanism is applied, if the message length is not a multiple of l. A message of length L can be padded by appending a 1 followed by the necessary number of zeros:

$$x_1 x_2 \dots x_L 10 \dots 0.$$

We write $m = m_1 || m_2 || \dots || m_N$, where each m_i is a block of length l.

- (ECB mode) Each block is encrypted separately using E_k, so that

  $$c_i = E_k(m_i) \text{ for } i = 1, 2, \dots, N, \text{ and } c = \mathcal{E}_k(m) = c_1 || c_2 || \dots || c_N.$$

  Decryption works in a similar way:

  $$m_i = E_k^{-1}(c_i) \text{ for } i = 1, 2, \dots, N, \text{ and } m = \mathcal{D}_k(c) = m_1 || m_2 || \dots || m_N.$$

- (Randomized CBC mode) An initialization vector $IV \overset{\$}{\leftarrow} \{0,1\}^l$ is chosen uniformly at random for each message. Then define

  $$\begin{aligned} c_0 &= IV, \\ c_i &= E_k(m_i \oplus c_{i-1}) \text{ for } i = 1, 2, \dots, N, \text{ and} \\ c &= \mathcal{E}_k(m) = c_0 || c_1 || c_2 || \dots || c_N. \end{aligned}$$

  Decryption is defined by

  $$\begin{aligned} m_i &= E_k^{-1}(c_i) \oplus c_{i-1} \text{ for } i = 1, 2, \dots, N \text{ and} \\ m &= \mathcal{D}_k(c) = m_1 || m_2 || \dots || m_N \end{aligned}$$

  (see Figure 2.10). ◊

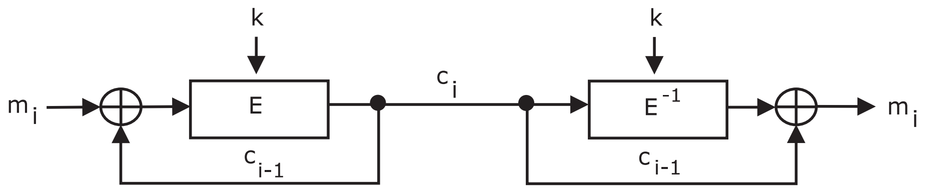


Figure 2.10. Encryption and decryption in CBC mode.

We can easily verify that the CBC mode has correct decryption:

$$E_k^{-1}(c_i) \oplus c_{i-1} = E_k^{-1}(E_k(m_i \oplus c_{i-1})) \oplus c_{i-1} = m_i \oplus c_{i-1} \oplus c_{i-1} = m_i.$$

The ECB mode has a straightforward definition, but the mode turns out to be insecure and should be avoided. The scheme is deterministic and thus cannot be CPA-secure. Neither is the scheme EAV-secure, since plaintext patterns are preserved. Suppose, for example, that $m = m_1 || m_2 || m_1 || m_2$; then the ciphertext has the same pattern: $c = c_1 || c_2 || c_1 || c_2$.

CBC is a popular mode, which can be CPA-secure when properly applied (see Theorem 2.52). The additional computational load, compared to the ECB mode, is very low. On the other hand, encryption in CBC mode cannot be parallelized.

Encryption schemes can also be based on a family of *functions* which are not necessarily bijective. The *counter mode* described below is often used in practice and effectively turns a block cipher into a stream cipher (see Chapter 6).

Definition 2.48. (Randomized CTR mode) Let

$$F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$$

be a family of functions. We define a variable-length symmetric encryption scheme based on F. The key generation algorithm $Gen(1^n)$ outputs a uniform random key $k \overset{\$}{\leftarrow} \{0,1\}^n$ of length n. A plaintext message m is split into blocks of length $l = l(n)$, where the last block can be shorter:

$$m = m_1 || m_2 || \dots || m_N .$$

A uniform random counter $ctr \overset{\$}{\leftarrow} \{0,1\}^l$ is chosen and viewed as an integer. For each block in the message, the counter is incremented (where addition is done modulo 2^l) and F_k is applied to the counter. The output is used as a keystream and the ciphertext is obtained by XORing plaintext and the keystream (see Figure 2.11):

$$c_i = F_k(ctr + i) \oplus m_i \text{ for } i = 1, 2, \dots, N \text{ and}$$
$$c = \mathcal{E}_k(m) = ctr || c_1 || c_2 || \dots || c_N .$$

Decryption is defined by XORing the ciphertext and the same keystream:

$$m_i = F_k(ctr + i) \oplus c_i \text{ for } i = 1, 2, \dots, N \text{ and}$$
$$m = \mathcal{D}_k(c) = m_1 || m_2 || \dots || m_N .$$

Only the first (most significant) bits of $F_k(ctr + N)$ are used for the XOR operation if the last block is shorter than l bits. $\Diamond$

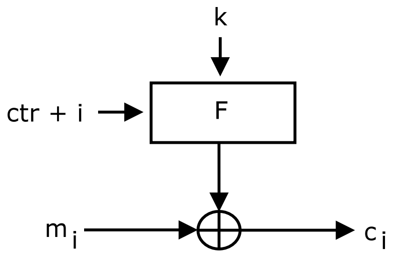


Figure 2.11. Encryption in CTR mode. Decryption is almost identical, with plaintext and ciphertext swapped.

The above CBC and CTR modes define randomized encryption schemes. The IV used in the CBC mode must be chosen uniformly, whereas the *ctr* value in the CTR mode can also be a *nonce* value (number used once). We note that IV and *ctr* values are not secret and (without encryption) become part of the ciphertext. The full ciphertext can only be recovered with the IV or *ctr* value.

Remark 2.49. The CTR mode turns a block cipher into a *synchronous stream cipher* (see Definition 6.1). The encrypted counter values are used as a *keystream*. For encryption and decryption, the plaintext and the ciphertext is XORed with the keystream. There are two other operation modes, the Cipher Feedback Mode (CFB) and the Output Feedback Mode (OFB) that also generate a keystream. These two modes and stream ciphers in general are discussed in Chapter 6. $\Diamond$

The following two Theorems 2.50 and 2.52 state that CTR and CBC modes achieve security under CPA if the scheme uses a pseudorandom family of functions or permutations.

Theorem 2.50. *If F is a pseudorandom function family, then the randomized CTR mode has indistinguishable encryptions under a chosen plaintext attack, i.e., the encryption scheme is IND-CPA secure.*

Proof. The assertion has a *proof by reduction*. Let A be an adversary in the CPA indistinguishability experiment (see Definition 2.24). We want to show that $\text{Adv}^{\text{ind-cpa}}(A)$ is negligible if F is a pseudorandom permutation.

We construct an algorithm B running in the prf experiment (see Definition 2.38), which uses A as a subroutine. Similarly as in the proof of Theorem 2.37, B takes the role of A's challenger in the CPA experiment. B generates a random bit $b \overset{\$}{\leftarrow} \{0,1\}$ and responds to A's encryption queries. So B needs to encrypt messages in CTR mode. Now, since B runs in the prf experiment, B has access to a function $f : \{0,1\}^l \to \{0,1\}^l$ which is either the pseudorandom function F_k or a random function. B uses f to encrypt messages of arbitrary length in CTR mode, and we denote the associated randomized encryption function by $\mathcal{E}_f$. If A sends the oracle query m, then B responds with $\mathcal{E}_f(m)$. During the experiment, A sends a pair (m_0, m_1) of plaintexts and B returns the challenge ciphertext $\mathcal{E}_f(m_b)$. Finally, A outputs b' and B observes whether or not A succeeds. The result is used to answer the challenge in the prf experiment: B outputs 1 if $b = b'$, and 0 otherwise.

The running time of B is polynomial and is given by the sum of A's running time and the time to encrypt the plaintexts chosen by A. Note that B's strategy is to observe A and to output 1, i.e., to guess that f is the pseudorandom function if A was successful.

Now we want to prove that B's advantage is closely related to A's advantage, so that $\text{Adv}^{\text{ind-cpa}}(A)$ is negligible if $\text{Adv}^{\text{prf}}(B)$ is negligible.

Since B is an adversary in the prf experiment, B does not know whether its challenger has chosen $f = \text{random}$ or $f = F_k$. We consider both cases:

(1) If f is a random function, then B succeeds, i.e., outputs 0 if and only if A fails. The sign is not relevant and the advantages of A and B are identical. The CTR mode encryption is similar to a one-time pad and the advantage of A and B is 0, unless the *counter values overlap*, i.e., a counter value used to compute the challenge ciphertext overlaps with at least one counter in the responses to A's queries. In this case, the adversary knows a keystream block and can easily answer the challenge. Let q be an upper bound on the number of blocks in A's chosen plaintexts m_0 and m_1, the number of queries and the number of blocks in the queries. Let ctr be the initial counter value used to compute the challenge ciphertext. For a single query and a chosen plaintext of at most q blocks, only initial counter values ctr' with $|ctr - ctr'| < q$ can result in an overlap. This inequality is satisfied for $2q - 1$ values of ctr'. If the number of queries is at most q, then an overlap occurs for less than $2q^2$ values. Since the initial counter values are chosen uniformly from $\{0,1\}^l$, we obtain

$$Pr[\text{Overlap}] < \frac{2q^2}{2^l}$$

(compare [**KL15**] and [**BR05**]). Since A runs in polynomial time, q is polynomial in n and $Pr[\text{Overlap}]$ is negligible. If an overlap occurs, then we use the trivial bound 1 for B's advantage. However, the probability of an overlap is less than $\frac{2q^2}{2^l}$, and otherwise the advantage of B is 0. We conclude that

$$|\underbrace{Pr[Out(B) = 0 \mid f = \text{random}\,] - Pr[Out(B) = 1 \mid f = \text{random}\,]}_{=:x}| < \frac{2q^2}{2^l}.$$

(2) If $f = F_k$, then B succeeds, i.e., outputs 1 if and only if A is successful. A runs in the CPA experiment with CTR mode encryption using the function F_k. Therefore,

$$\text{Adv}^{\text{ind-cpa}}(A) = |\underbrace{Pr[Out(B) = 1 \mid f = F_k] - Pr[Out(B) = 0 \mid f = F_k]}_{=:y}|.$$

A uniform random bit, chosen by B's challenger in the prf experiment, determines whether f is a random function or $f = F_k$. Hence

$$Pr[f = \text{random}\,] = Pr[f = F_k] = \frac{1}{2}\,.$$

The definition of B's advantage in the prf experiment and the definition of x and y in (1) and (2) yield

$$\text{Adv}^{\text{prf}}(B) = \left|\frac{1}{2}x + \frac{1}{2}y\right|.$$

Combining (1) and (2), we obtain

$$\text{Adv}^{\text{ind-cpa}}(A) = |y| = |y + x - x| \leq |x + y| + |x| < 2\ \text{Adv}^{\text{prf}}(B) + \frac{2q^2}{2^l}.$$

F is a pseudorandom function family, and hence the advantage $\mathrm{Adv}^{\mathrm{prf}}(B)$ is negligible. Furthermore, $\frac{1}{2^l}$ is negligible in l as well as in n, and $\frac{2q^2}{2^l}$ is negligible in n. It follows that $\mathrm{Adv}^{\mathrm{ind-cpa}}(A)$ is negligible, which completes the proof by reduction. $\square$

Remark 2.51. The security guarantee provided by this type of proof can easily be misunderstood, and there have been some controversies about this topic (see [**KM07**] and the subsequent discussion).

- *Is encryption in CTR mode unconditionally secure?*
  No, since the CPA security depends on the security of the underlying pseudorandom function family. The scheme is secure under the condition that the function family is pseudorandom. In other words, one has to assume that no polynomial-time algorithm can distinguish between the function family and a random function, or at least that no one can find such an algorithm.
- *If pseudorandomness is given, is the CTR mode then secure against all polynomial-time attacks?*
  No, since we made assumptions about the capabilities of adversaries. They are only looking at plaintexts and ciphertexts and have no information about the secret key. Side-channel attacks, where an adversary analyzes the power consumption or the timing of an implementation are not considered. Furthermore, one assumes that the secret key is uniformly random. This is not necessarily true in practice, for example if the key is derived from a password.
- *If all assumptions are satisfied, does the CTR mode have a concrete security guarantee?*
  The CTR mode is asymptotically CPA-secure. Concrete security depends on the security parameter, the security of F and the number and length of queries made by an adversary.

  We given an example and estimate the concrete advantage of an adversary A given in the proof of the above Theorem. Suppose that $l = 128$, A makes $q = 2^{32}$ queries and the advantage of the corresponding adversary B in the prf experiment is $\mathrm{Adv}^{\mathrm{prf}}(B) = 2^{-64}$. We assume that the number of blocks in A's chosen plaintexts and in his queries is less than q. Then

  $$\mathrm{Adv}^{\mathrm{ind-cpa}}(A) \leq 2\ \mathrm{Adv}^{\mathrm{prf}}(B) + \frac{2q^2}{2^l} = 2^{-62}.$$

  The concrete security guarantee under the above assumptions is quite strong since the CPA advantage is very small. $\Diamond$

The randomized CBC mode also has a security guarantee, and we refer to [**KL15**] or [**BR05**] for a proof by reduction.

Theorem 2.52. *If E is a pseudorandom permutation, then the randomized CBC mode is IND-CPA secure.*

Remark 2.53. It is easy to see that neither the CBC nor the CTR modes yield a CCA2-secure encryption scheme. Security against chosen ciphertext attacks requires *non-malleability*; a controlled modification of the ciphertext should be impossible and the decryption of a modified ciphertext should not be related to the original plaintext. For example, let's consider the CTR mode and suppose a single bit of the ciphertext is flipped. Then only the corresponding plaintext bit changes, so that the CTR mode is clearly malleable.

CCA2 security of encryption schemes can be achieved by incorporating a *message authentication code*. This is, for example, implemented by the Galois Counter Mode (GCM), which is explained in Section 8.4.

2.11. Summary

- Encryption schemes are defined by plaintext, ciphertext and key spaces as well as key generation, encryption and decryption algorithms. The encryption algorithm can be probabilistic.
- Perfect secrecy is a very strong requirement, but the one-time pad is perfectly secret. The key must be at least as long as the plaintext and cannot be re-used.
- Security against eavesdropping (EAV), chosen plaintext attacks (CPA) or adaptive chosen ciphertext attacks (CCA2) are common security goals of encryption schemes. The security definition is based on experiments and the performance of adversaries.
- Pseudorandom generators (prg) are deterministic algorithms. Polynomial-time adversaries cannot distinguish the output of a prg from a random string if the seed (or key) is secret.
- Pseudorandom function families (prf) and permutations appear to be random transformations to polynomial-time adversaries if the key is secret.
- Block ciphers are keyed permutations. The combination of a block cipher and an operation mode defines a variable-length encryption scheme.
- The randomized CBC and CTR modes give CPA-secure encryption schemes if the underlying block cipher is a pseudorandom permutation.

Exercises

1. Show that the Vigenère cipher is perfectly secure if the key is randomly chosen, it is only used once and the plaintext has the same length as the key.
2. Find reasons for Kerkhoff's principle and discuss possible counter-arguments.
3. Show that the one-time pad is not perfectly secret if a key is used twice.
4. Let $\mathcal{M}$ be the plaintext space and $\mathcal{K}$ the key space of a perfectly secure encryption scheme. Show that $|\mathcal{K}| \geq |\mathcal{M}|$.

Hint: Suppose $\mathcal{E}_k(m_0) = c$. How many different plaintext-key pairs give the ciphertext c?

5. Is a bit permutation of block length n perfectly secure if it is used only once to encrypt a string of length n?
6. Show the formulas for the advantage of adversaries in Remark 2.16.
7. Explain the differences in the definitions of EAV-secure and IND-CPA secure encryption schemes.
8. Prove that a perfectly secure scheme is EAV-secure. Show that $\mathrm{Adv}^{\mathrm{eav}}(A)$ is 0 for any adversary A. Why is perfect security much stronger than EAV security?
9. Does the Vigenère cipher define an EAV-secure encryption scheme?
10. Suppose we want to expand a fixed output-length pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^{n+1}$ by one extra bit. Which of the following generators $G^+ : \{0,1\}^n \to \{0,1\}^{n+2}$ could be pseudorandom?
    (a) $G(s) = (y_1 \| y_2 \| \dots \| y_{n+1})$, $G^+(s) = (y_1 \| y_2 \| \dots \| y_{n+1} \| \, y_1 \oplus y_2 \cdots \oplus y_{n+1})$.
    (b) $s_0 = s$, for $i = 1, \dots, n+2$: $(s_i \,\|\, y_i) = G(s_{i-1})$. $G^+(s) = (y_1 \| y_2 \| \dots \| y_{n+2})$.
11. Suppose G is a pseudorandom generator with fixed output-length and associated encryption scheme defined by $\mathcal{E}_k(m) = m \oplus G(k)$. Show that this scheme is not EAV-secure for multiple encryptions.
12. Prove that a pseudorandom generator is unpredictable in polynomial time, i.e., passes the next-bit test.
13. Explain why a malleable encryption scheme cannot be CCA2-secure.
14. Let F be a family of bit permutations. Is F a pseudorandom permutation?
15. Show that a block cipher in ECB mode is not EAV-secure.
16. Consider a block cipher in CBC mode. Suppose that the IV is initially set to 0 and then incremented for every new encryption. Can this variant of the CBC mode be CPA-secure?
17. Show that a block cipher in CTR is not secure against ciphertext-only attacks, if the counter is re-used.
18. Can an encryption scheme that is based on a block cipher be perfectly secure?
19. Let E be a block cipher of block length 4 and suppose that $E_k(b_1 b_2 b_3 b_4) = (b_2 b_3 b_4 b_1)$. Encrypt $m = 1011\ 0001\ 0100$ and decrypt the ciphertext with the following operation modes:
    (a) ECB mode,
    (b) CBC mode with $IV = 1010$,
    (c) CTR mode with $ctr = 1010$.

20. Consider a block cipher in CBC mode. The ciphertext is sent to a receiver. What are the consequences, if:
    (a) the receiver misses the initialization vector (IV), or
    (b) a single ciphertext block is changed due to transmission errors, or
    (c) a ciphertext bit is flipped by an adversary during the transmission, or
    (d) a bit error occurs during the ciphering operation ?
21. Suppose a block cipher of block length 128 in CTR mode is used to encrypt 300 bits of plaintext. Which bits cannot be correctly decrypted if one of the following errors occurs?
    (a) The first bit of the counter value is flipped during transmission.
    (b) The first ciphertext bit is flipped.
    (c) The first ciphertext block c_1 is changed due to transmission errors.
    (d) The last ciphertext bit is flipped.
22. Compare ECB, CBC and CTR modes with respect to message expansion, error propagation, pre-computations and parallelization of encryption and decryption.
23. Explain why neither CBC nor CTR modes achieve CCA2 security.

Chapter 3

Elementary Number Theory

This chapter covers several topics from elementary number theory. Section 3.1 deals with integers, factorizations, prime numbers and the Euclidean algorithm. In Section 3.2, we discuss residue classes and modulo operations. The modular exponentiation and the associated algorithms play an important role in several cryptographic schemes and are dealt with in Section 3.3.

We refer to [**Ros12**] and [**Sho09**] for detailed expositions of the modular arithmetic and the elementary number theory used in cryptography.

3.1. Integers

The set of *integers* $\{\dots, -2, -1, 0, 1, 2, 3, \dots\}$ is denoted by $\mathbb{Z}$. Integer numbers can be added, subtracted and multiplied. The fractions $\frac{a}{b}$ with $a, b \in \mathbb{Z}$ and $b \neq 0$ define the *rational numbers* $\mathbb{Q}$. We say that b *divides* a, a *is divisible by* b or a *is a multiple of* b, if $\frac{a}{b} \in \mathbb{Z}$ and we write $b \mid a$. Otherwise, we write $b \nmid a$.

Example 3.1. $2 \mid 224236$, but $2 \nmid -71$. One has $1 \mid n$ and $(-1) \mid n$ for all $n \in \mathbb{Z}$. Although $7 \mid 14$, note that $14 \nmid 7$. ◊

The result of an integer division $a : b$ for $a,\ b \in \mathbb{Z}$, $b \neq 0$ consists of an integer quotient $q = \lfloor \frac{a}{b} \rfloor \in \mathbb{Z}$ and a remainder $r \in \mathbb{Z}$ such that $a = bq + r$ and $0 \leq r < b$ holds. Then $a \equiv r \mod b$ since a and r are congruent modulo b, i.e., elements of the same residue class. If b divides a, then the remainder is 0 and $a \equiv 0 \mod b$.

Example 3.2. (1) $23 : 3 = 7$ remainder 2 and $23 = 3 \cdot 7 + 2$. Thus $23 \equiv 2 \mod 3$.

(2) $(-17) : 5 = -4$ remainder 3 and $-17 = 5 \cdot (-4) + 3$. Rather, one might expect that the integer quotient is -3 in this example, but then the remainder would be negative. Hence $-17 \equiv 3 \mod 5$, although $-17 \equiv -2 \mod 5$ is also correct. $\diamondsuit$

These elementary operations have efficient algorithms:

Proposition 3.3. *Let $a, b \in \mathbb{Z}$. Then $a + b$ and $a - b$ can be computed in time $O(\max(\text{size}(a), \text{size}(b)))$ and the product $a \cdot b$ in time $O(\text{size}(a)\ \text{size}(b))$. We can compute the integer quotient q and the remainder r of the integer division $a : b$ in time $O(\text{size}(b)\ \text{size}(q))$.*

Proof. Use a base representation of a and b (and a sign) and analyze the complexity of the well-known "paper-and-pencil" methods. $\square$

Definition 3.4. A positive integer $p \in \mathbb{N}$ with $p \geq 2$ is called *prime* if p is only divisible by ± 1 and $\pm p$. Integers $n \geq 2$ that are not prime numbers are called *composite*. $\diamondsuit$

The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, and large prime numbers play an important role in cryptography. The following famous theorem describes the asymptotic density of primes (see [**Edw74**]):

Theorem 3.5. (*Prime Number Theorem*) *Let $\pi(x)$ be the number of primes less than x. Then*

$$\lim_{x \to \infty} \frac{\pi(x)}{\left(\frac{x}{\ln(x)}\right)} = 1.$$

Therefore, the prime density $\frac{\pi(x)}{x}$ is asymptotically $\frac{1}{\ln(x)}$.

Example 3.6. Let $x = 2^{2048}$. The prime density $\frac{\pi(x)}{x}$ is approximately $\frac{1}{\ln(x)} = \frac{1}{2048 \ln(2)}$ $\approx \frac{1}{1386}$. Hence the prime density of *odd integers* with at most 2048 binary digits is $\approx \frac{2}{1386} \approx 0.0014$.

Theorem 3.7. (*Fundamental Theorem of Arithmetic*) *Every nonzero integer can be decomposed into a product of primes and a sign (factor 1 or -1). The decomposition is unique up to the order of the factors.*

Example 3.8. $-60 = -2^2 \cdot 3 \cdot 5$.

Example 3.9. We can use SageMath to check the primality or to obtain the factorization of an integer.

```
sage: is_prime(267865461)
False
sage: factor(267865461)
3^5 * 337 * 3271
```

Primality testing for large numbers can be time-consuming, although a polynomial-time algorithm (the AKS primality test [**AKS04**]) is known. In practice, probabilistic algorithms are used (see Section 9.4).

```
sage: is_pseudoprime(2^1279-1)
True
```

The largest prime number known at the time of writing is the *Mersenne prime*

$$2^{82589933} - 1.$$

This prime number has more than 24 million decimal digits. The primality of Mersenne numbers $2^n - 1$ is tested with the Lucas-Lehmer test, which we do not deal with in this book.

Definition 3.10. Let a and b be nonzero integers. Then the greatest positive integer that divides both $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ is called the *greatest common divisor* of a and b and is denoted by $\gcd(a, b)$. We say that a and b are *relatively prime* if $\gcd(a, b) = 1$. ♢

Example 3.11. (1) $\gcd(12, 140) = 4$, since $12 = 2^2 \cdot 3$ and $140 = 2^2 \cdot 5 \cdot 7$.

(2) 9 and 26 are relatively prime since $\gcd(3^2, 2 \cdot 13) = 1$, although neither 9 nor 26 are prime numbers.

(3) If p is a prime number, then all integers $1 \leq n < p$ are relatively prime to p.

(4) An example using SageMath:

```
sage: gcd(267865461,236749299182338)
3271
```

Proposition 3.12. *Let a and b be nonzero integer numbers. Then there exist x, $y \in \mathbb{Z}$ such that*

$$\gcd(a, b) = ax + by\,. \qquad ♢$$

The greatest common divisor and the above numbers x and y can be efficiently computed with the *Extended Euclidean Algorithm* (see Algorithm 3.1). The algorithm plays an important role in elementary number theory and in cryptography.

Algorithm 3.1 Extended Euclidean Algorithm

```
Input: $a,\ b \in \mathbb{N}$
Output: $\gcd(a,b),\ x,\ y \in \mathbb{Z}$ such that $\gcd(a,b) = ax + by$
Initialisation: $x_0 = 1, x_1 = 0, y_0 = 0, y_1 = 1, sign = 1$
while $b \neq 0$ do
    $r = a \bmod b$ // remainder of the integer division $a : b$
    $q = a/b$ // integer quotient
    $a = b$
    $b = r$
    $xx = x_1$
    $yy = y_1$
    $x_1 = q \cdot x_1 + x_0$
    $y_1 = q \cdot y_1 + y_0$
    $x_0 = xx$
    $y_0 = yy$
    $sign = -sign$
end while
$x = sign \cdot x0$
$y = -sign \cdot y0$
$gcd = a$
return gcd, $x$, $y$
```

Example 3.13. We compute gcd(845, 117) (see Table 3.1). We have

Table 3.1. Computation of $\gcd(845, 117) = 13$.

$845 : 117 = 7$ rem. 26	$845 = 7 \cdot 117 + 26$	$26 = 845 - 7 \cdot 117$
$117 : 26 = 4$ rem. 13	$117 = 4 \cdot 26 + 13$	$13 = 117 - 4 \cdot 26$
$26 : 13 = 2$ rem. 0	$26 = 2 \cdot 13 + 0$	

$$\mathbf{13} = 117 - 4 \cdot 26 = 117 - 4 \cdot (845 - 7 \cdot 117) = \mathbf{-4} \cdot 845 + \mathbf{29} \cdot 117.$$

Hence $\gcd(845, 117) = 13$, $x = -4$, $y = 29$. ◊

We can assume that the input parameters of the algorithm satisfy $a > b$, since otherwise the first iteration of the algorithm swaps a and b.

The Extended Euclidean Algorithm is very efficient:

Proposition 3.14. *The running time of the Extended Euclidean Algorithm on input $a, b \in \mathbb{N}$ is $O(\text{size}(a)\ \text{size}(b))$.* ◊

We refer to [**Sho09**] for a detailed proof of this statement. Suppose that $a > b > 0$. Each step of the **while** loop has running time $O(\text{size}(a)\ \text{size}(q))$, where q

is the current integer quotient. Since the product of all quotients q is less than or equal to a, the algorithm runs in time $O(\text{size}(a)^2)$. A more refined argument shows that the running time is $O(\text{size}(a)\ \text{size}(b))$.

3.2. Congruences

Congruences and residue classes modulo n were already dealt with in Chapter 1 (see Example 1.21). Let $n \geq 2$ be a positive integer. Recall the definition of the following equivalence relation on $\mathbb{Z}$:

$$R_n = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x - y \in n\mathbb{Z}\}.$$

Hence $(x, y) \in R_n$ if the difference $x - y$ is divisible by n. Equivalent elements x and y are called *congruent modulo n* and we write $\overline{x} = \overline{y}$ or $x \equiv y \mod n$. The set of equivalence classes is denoted by $\mathbb{Z}_n$ or $\mathbb{Z}/(n)$ and contains n elements. $\mathbb{Z}_n$ is also called the set of *residue classes* mod n or *integers mod n* (see Figure 3.1).

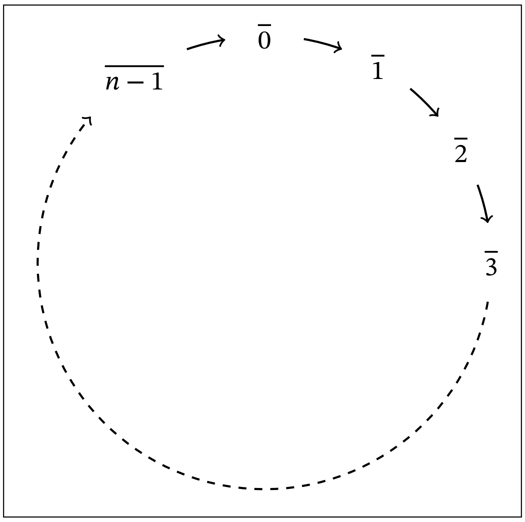


Figure 3.1. Residue classes modulo n.

Two integers x and y are congruent mod n if they have the same *remainder* when they are divided by n. Obviously, if $x = q_1 n + r_1$ and $y = q_2 n + r_2$ with $r_1,\ r_2 \in \{0,\ 1,\ \dots,\ n-1\}$, then $x - y = (q_1 - q_2)n + (r_1 - r_2)$. Hence x and y are congruent mod n if and only if $r_1 - r_2 = 0$. In other words, $x \equiv y \mod n$ holds if the integer divisions $x : n$ and $y : n$ have the same remainder.

Note that an integer $x \in \mathbb{Z}$ is only one *representative* of the residue class $\overline{x} \in \mathbb{Z}_n$, which contains infinitely many congruent elements. The *standard representatives* of $\mathbb{Z}_n$ are $0,\ 1,\ \dots,\ n-1$, but other representatives are also permitted. Elements in $\mathbb{Z}_n$ can be added, subtracted and multiplied by choosing an arbitrary representative in $\mathbb{Z}$. It is often reasonable to choose the standard representative before and after each step of a computation in order to reduce the size of numbers.

Example 3.15. (1) $234577 \cdot 2328374 \cdot 2837289374 \mod 3 \equiv 1 \cdot 2 \cdot 2 \equiv 1 \mod 3$.

(2) An example using SageMath:

```
sage: mod (782637846,8927289)
5963703
```

(3) How would you compute $782637846 \mod 8927289$ with a simple pocket calculator? The real division $782637846 : 8927289$ gives approximately 87.668, and so the integer quotient is 87. We compute $782637846 - 87 \cdot 8927289$ and get the remainder 5963703. Alternatively, we multiply the fractional part 0.668 by 8927289 and also obtain 5963703, up to a small rounding error. ◊

Modular addition, subtraction and multiplication is straightforward using any integer representative, and then showing that the result does not depend on the chosen representative. The *division of residue classes* is more tricky. Since $\frac{b}{a} \mod n$ might be confused with the rational number $\frac{b}{a}$, we rather write

$$(b \mod n) \cdot (a \mod n)^{-1}$$

if $(a \mod n)^{-1}$ exists. A nonzero integer a is invertible modulo n if the equation

$$ax \equiv 1 \mod n$$

has a solution $x \in \mathbb{Z}$. In this case, x is a representative of $(a \mod n)^{-1}$. One can show that the modular inverse, if it exists, is uniquely determined.

Example 3.16. Does the residue class $(3 \mod 10)^{-1}$ exist ? In other words, is 3 invertible modulo 10 ? Since the equation

$$3x \equiv 1 \mod 10$$

is solved by $x = 7$, we find that $(3 \mod 10)^{-1} \equiv 7 \mod 10$. Note that the result is neither $\frac{1}{3} \in \mathbb{Q}$ nor the integer quotient or remainder of $1 : 3$. ◊

When does the multiplicative inverse modulo n exist and how can we efficiently compute it?

Proposition 3.17. *Let $n \geq 2$ be a positive integer and $a \in \mathbb{Z}$ a nonzero integer. Then $a \mod n$ has a multiplicative inverse if and only if $\gcd(a,n) = 1$. A representative of $(a \mod n)^{-1}$ can be efficiently computed using the Extended Euclidean Algorithm.*

Proof. Running the Extended Euclidean Algorithm on input a and n gives the output integers $\gcd(a,n)$, x and y such that

$$\gcd(a,n) = ax + ny.$$

If $\gcd(a, n) = 1$, then $1 \equiv ax \mod n$, and so x is a representative of $(a \mod n)^{-1}$. Conversely, if a is invertible modulo n, then $ax = 1 + ny \Leftrightarrow ax - ny = 1$ for some $x, y \in \mathbb{Z}$. This implies $\gcd(a, n) \mid 1$, which is only possible if $\gcd(a, n) = 1$. $\square$

Definition 3.18. The invertible integers modulo n are called *units* mod n. The subset of units of $\mathbb{Z}_n$ is denoted by $\mathbb{Z}_n^*$. Euler's φ-function (or ϕ-function) is defined by the cardinality of the units mod n, i.e., $\varphi(n) = |\mathbb{Z}_n^*|$.

Example 3.19. (1) $\mathbb{Z}_{10}^* = \{\overline{1}, \overline{3}, \overline{7}, \overline{9}\}$ and $\varphi(10) = 4$. The inverse elements are as follows:

$$\overline{1}^{-1} = \overline{1},\ \overline{3}^{-1} = \overline{7},\ \overline{7}^{-1} = \overline{3},\ \overline{9}^{-1} = \overline{9}.$$

The inverses can be computed with the Extended Euclidean Algorithm. If we take the input values $a = 3$ and $n = 10$, then we obtain $\gcd(3, 10) = 1$, $x = -3$ and $y = 1$, satisfying the equation $1 = 3 \cdot (-3) + 10 \cdot 1$. Hence $(3 \mod 10)^{-1} \equiv -3 \equiv 7 \mod 10$.

(2) Let p be a prime number; then $\mathbb{Z}_p^* = \{\overline{1}, \overline{2}, \dots, \overline{p-1}\}$ and $\varphi(p) = p - 1$.

3.3. Modular Exponentiation

The exponentiation of residue classes plays an important role in several cryptographic algorithms. The exponent can be very large and may have in practice more than 2000 binary digits. A straightforward computation of modular powers is inefficient or even completely impossible.

Below, we explain the *fast exponentiation* algorithm for residue classes. The algorithm is based on the following observation: let $a = 2^k$ be a power of 2. Then

$$x^a \mod n = ((((x^2 \mod n)^2 \mod n)^2 \mod n)^2 \dots)^2 \mod n,$$

where the squaring is iterated k times. For example, $x^{(2^{1024})} \mod n$ can be computed with only 1024 modular squaring operations, even though 2^{1024} is a huge number. After each squaring, one takes the result modulo n in order to reduce the size of integers.

Warning 3.20. x^{2^k} is not the same as x^{2k}. In fact, $x^{2^k} = x^{(2^k)}$, and this is different from $(x^2)^k = x^{2k}$. $\Diamond$

If the exponent is not a power of 2, then it can still be written as a sum of powers of 2. This gives a product of factors x^{2^k}. The binary representation of the exponent determines whether or not a factor x^{2^k} is present in the product.

Example 3.21. $6^{41} \mod 59$. We have $41 = 2^5 + 2^3 + 2^0$ and compute the following sequence of squares:

$$\begin{aligned} 6^2 &\equiv 36 \mod 59, \\ 6^4 &\equiv 36^2 \equiv 57 \mod 59, \\ 6^8 &\equiv 57^2 \equiv 4 \mod 59, \\ 6^{16} &\equiv 4^2 \equiv 16 \mod 59, \\ 6^{32} &\equiv 16^2 \equiv 20 \mod 59. \end{aligned}$$

The intermediate result $6^8 \equiv 4$ should be stored when computing 6^{32}. Then

$$6^{41} = 6^{32} \cdot 6^8 \cdot 6 \equiv 20 \cdot 4 \cdot 6 \equiv 8 \mod 59.$$

Hence five quadratures and two modular multiplications are needed.

SageMath uses the function `power_mod`:

```
sage: power_mod(6,41,59)
8
```

Proposition 3.22. *Let $x, a, n \in \mathbb{N}$. The computation of the modular power $x^a \mod n$ requires at most* size(n) − 1 *quadratures and* size(n) − 1 *multiplications modulo n, if the fast exponentiaton algorithm is used. The total number of modular multiplications is bounded above by* 2 size(n). *The complexity of modular exponentiations is* $O(\text{size}(n)^3)$. ◊

Since the complexity of a single modular multiplication is $O(\text{size}(n)^2)$, the complexity of a modular exponentiation is $O(\text{size}(n)^3)$. This is polynomial in size(n), but significantly slower than algorithms with linear or quadratic running time. Therefore, one tries to avoid too many exponentiations in practice. Algorithms that use modular powers (for example RSA and Diffie-Hellman) are not applied to bulk data since they are too slow.

Warning 3.23. When computing $x^a \mod n$, the *exponent must not be reduced mod n*. For example $2^6 = 64 \equiv 4 \mod 5$, which is different from $2^1 = 2 \mod 5$. However, we will later see (Proposition 4.16) that a *reduction mod $\varphi(n)$* is possible (i.e., mod 4 in this example so that $2^6 \equiv 2^2 = 4 \mod 5$) and helps to reduce the size of the exponent. Furthermore, a reduction of the *base x mod n* is allowed. For example, $6^6 \equiv 1^6 = 1 \mod 5$. ◊

The fast exponentiation can be slightly optimized with the *square-and-multiply* algorithm (see Algorithm 3.2).

Example 3.24. $6^{41} = ((((6^2)^2 \cdot 6)^2)^2)^2 \cdot 6$. The computation is a sequence of squarings (SQ) and multiplications (MULT): SQ, SQ, MULT, SQ, SQ, SQ, MULT. As above, we need 5 quadratures and 2 multiplications, but in a different order. The exact order depends on the binary representation of the exponent. An advantage of this algorithm is that intermediate results do not need to be stored.

Algorithm 3.2 Square-and-Multiply Algorithm

```
Input: Base $x \in \mathbb{N}$ and exponent $a = \sum_{i=0}^{s} h_i 2^i$ with $h_i \in \{0,1\}$ and $a \geq 2$.
Output: $x^a \bmod n$
Initialisation: $r = x$
1: for $i = s-1$ downto 0 do
2:     $r = r^2 \bmod n$
3:     if $h_i = 1$ then
4:         $r = r \cdot x \bmod n$
5:     end if
6: end for
7: return $r$
```

3.4. Summary

- Every nonzero integer can be decomposed into a product of prime numbers and a sign.
- The Extended Euclidean Algorithm takes two integers as input and outputs their greatest common divisor and two other useful integers.
- There are n different residue classes modulo n. They are also called integers modulo n.
- Residue classes can be added, subtracted and multiplied. An integer can be inverted modulo n if the greatest common divisor of that integer and n is equal to 1.
- There exist efficient algorithms for the elementary arithmetic operations (addition, multiplication, division, etc.) on integers and residue classes.
- Fast exponentiation and square-and-multiply are efficient algorithms to compute a modular power.

Exercises

1. Which of the following residue classes are identical in $\mathbb{Z}_{26}$:

$$\overline{0},\ \overline{3},\ \overline{-49},\ \overline{49},\ \overline{104}?$$

2. Enumerate the elements of $\mathbb{Z}_{22}^*$ and give $\varphi(22)$. Find the inverse of each element in $\mathbb{Z}_{22}^*$.
3. Perform some elementary modular operations with SageMath.
   Let $n = 123456789012345$, $a = 5377543210987654321$ and $b = 12345678914335$.
   (a) Find the prime factor decomposition of n, a and b. Use the `factor` command.
   (b) Compute $a + b \bmod n$, $ab \bmod n$, $a^b \bmod n$, using `mod(..,..)` and `power_mod(..,..,..)`.
   (c) Are a or b invertible modulo n ? Why or why not? Compute $(a \bmod n)^{-1}$ or $(b \bmod n)^{-1}$, using `mod(1/..,..)`.
   (d) Are a and b relatively prime? Why or why not?
4. Run the Extended Euclidean Algorithm on input $a = 1234$ and $b = 6789$.
5. Use the Extended Euclidean Algorithm to compute the multiplicative inverse of $\overline{32} \in \mathbb{Z}_{897}^*$.
6. Let p be a prime number and $m \in \mathbb{N}$. Find $\varphi(2p)$, $\varphi(2^m)$ and $\varphi(p^m)$.
7. Write a function which examines the primality of all *Mersenne numbers*

$$M_n = 2^n - 1$$

   for $n < 2000$. Use the SageMath function `is_pseudoprime(...)`.
8. This exercise explores the plain RSA encryption scheme (see Section 9.2).
   (a) Let p and q be the largest Mersenne prime numbers with less than 2000 binary digits (see Exercise 7).
   Compute $n = pq$ and $\varphi(n) = (p - 1)(q - 1)$.
   (b) Find all integers $1 < e < 100$ such that $\gcd(e, \varphi(n)) = 1$. We will later see that (e, n) can be used as a public RSA key. Now choose $e > 1$ as small as possible such that e and $\varphi(n)$ are relatively prime.
   (c) Compute the private RSA key $d = (e \bmod \varphi(n))^{-1}$.
   (d) Suppose that $m = 2^{1500} + 2^{500} + 1$ is a given plaintext. Compute the RSA ciphertext $c = m^e \bmod n$.
   (e) Decrypt the ciphertext by computing $c^d \bmod n$ and compare the result with the plaintext m.

   Tip: SageMath distinguishes between residue classes and integers. Use `ZZ(d)` to transform d from a residue class into an integer.
9. Write a SageMath function which generates 100,000 odd random integers less than 2^{1024}, checks their primality and counts the number of primes. Use the SageMath function `ZZ.random_element(N)` to generate a positive random integer less than N. Compare the number of primes with the expected number using the Prime Number Theorem.
   Tip: An odd random integer less than 2^{1024} can be generated by
   `2*(ZZ.random_element(`2^{1023}`))+1`.
   Test the primality with the function `is_pseudoprime`.

10. Compute $2^{55} \mod 61$
    (a) using fast exponentiation and
    (b) using the square-and-multiply algorithm.
    How many modular squarings and multiplications are necessary?
11. Determine the maximum number of modular squarings and multiplications that are needed to compute $x^k \mod n$ if $\text{size}(n) = \text{size}(k) = 2048$.

Chapter 4

Algebraic Structures

Modern cryptography uses not only discrete mathematics and elementary number theory, but also algebraic structures such as abelian groups, polynomial rings, quotient rings and finite fields. Understanding finite abelian groups and finite fields is crucial, and these algebraic topics should not be underestimated.

Section 4.1 deals with groups and, in particular, finite abelian groups. We will see that finite abelian groups are products of cyclic groups. Rings and fields are discussed in Section 4.2. Finite fields are used in many cryptographic constructions, and in Section 4.3, we construct fields with p^n elements. We add a recapitulation of linear and affine maps in Section 4.4.

The contents of this chapter can be found in any textbook on abstract algebra, and we refer the reader for example to [**Sho09**].

4.1. Groups

Groups are among the most fundamental mathematical structures. A group is a set with a binary operation which satisfies several properties.

Definition 4.1. A *group* G is a set together with a law of composition

$$\circ : G \times G \to G$$

such that the following properties are satisfied:

- For all $a, b, c \in G$ one has $(a \circ b) \circ c = a \circ (b \circ c)$ (associative law).
- There is an *identity* element $e \in G$ such that $e \circ g = g \circ e = g$ for all $g \in G$ (identity element).

- For every $g \in G$ there is an *inverse* element $x \in G$ with $g \circ x = x \circ g = e$ (inverse elements).

The group is called *abelian* or *commutative* if for all $a, b \in G$, one has $a \circ b = b \circ a$ (commutative law). ♢

The above definition uses $\circ$ for the composition of elements. In our applications, the composition is either addition or multiplication, and we write $+$ (plus) or $\cdot$ (dot). The identity element is denoted by 0 (additive case) or 1 (multiplicative case). Accordingly, the inverse element of g is denoted by $-g$ in the additive case and by g^{-1} in the multiplicative case.

Example 4.2. (1) $(\mathbb{Z}, +)$ is an additive abelian group.

(2) $(\mathbb{R} \setminus \{0\}, \cdot)$ is a multiplicative abelian group.

(3) $(\mathbb{Z}_n, +)$ is an additive abelian group with n elements.

(4) $(\mathbb{Z}_n^*, \cdot)$ is a multiplicative abelian group with $\varphi(n)$ elements (see Definition 3.18). We removed all non-units from $\mathbb{Z}_n$ to ensure that all elements are invertible. Note that the inverses are again residue classes, not rational numbers!

(5) Let p be a prime. Then $(\mathbb{Z}_p^*, \cdot)$ is a multiplicative abelian group with $p-1$ elements (see Example 3.19).

(6) Let G_1 and G_2 be groups; then the direct product $G_1 \times G_2$ is also a group, consisting of all tuples (g_1, g_2) with $g_1 \in G_1$ and $g_2 \in G_2$. The group operation is defined component-wise. More generally, one has the product groups

$$G_1 \times \cdots \times G_n \text{ and } G^n = G \times \cdots \times G.$$

Remark 4.3. Examples of *non-commutative groups* include the group of permutations of a set S (with composition of mappings) and the group of invertible matrices (with matrix multiplication). The structure of these groups can be very complicated. ♢

We want to relate different groups and consider maps that respect the group structure.

Definition 4.4. Let $f : G_1 \to G_2$ be a map between two groups G_1, G_2. Then f is called a *group homomorphism* if $f(g \circ g') = f(g) \circ f(g')$ for all $g, g' \in G_1$. A bijective group homomorphism is called an *isomorphism*. Then we say that G_1 is *isomorphic* to G_2 and write $G_1 \cong G_2$. ♢

It is easy to show that the inverse map f^{-1} of an isomorphism f is also a group homomorphism and hence an isomorphism.

Warning 4.5. A bijection between two groups does not imply that they are isomorphic! For example, there is a bijection between $\mathbb{Z}_2 \times \mathbb{Z}_2$ and $\mathbb{Z}_4$ (since both groups have 4 elements), but they are not isomorphic (see Example 4.28 below).

Example 4.6. (1) The natural projection map $f : \mathbb{Z} \to \mathbb{Z}_n$ with $f(k) = k \mod n$ is a group homomorphism, since

$$f(k_1 + k_2) = (k_1 + k_2) \mod n = (k_1 \mod n) + (k_2 \mod n).$$

Obviously, f is not injective and therefore not an isomorphism.

(2) The reverse map $f : \mathbb{Z}_n \to \mathbb{Z}$ with $f(\overline{k}) = k$, where $k \in \{0, 1, \dots, n-1\}$ is the standard representative, is *not* a homomorphism, since for example

$$f(\overline{1}) + f(\overline{n-1}) = 1 + (n-1) = n,$$

but $f(\overline{1} + \overline{n-1}) = f(\overline{0}) = 0$.

(3) Let $G_1 = (\mathbb{Z}_4, +) = \{\overline{0}, \overline{1}, \overline{2}, \overline{3}\}$ and $G_2 = (\mathbb{Z}_5^*, \cdot) = \{\overline{1}, \overline{2}, \overline{3}, \overline{4}\}$. The map

$$f : G_1 \to G_2,\ f(k \mod 4) = 2^k \mod 5$$

is well defined, since $2^4 \equiv 16 \equiv 1 \mod 5$ so that the result does not depend on a representative of k modulo 4. It defines a homomorphism, since

$$f((k_1 \mod 4) + (k_2 \mod 4)) = 2^{k_1+k_2} \mod 5 = (2^{k_1} \mod 5) \cdot (2^{k_2} \mod 5).$$

The explicit mapping is given by $f(\overline{0}) = \overline{1}$, $f(\overline{1}) = \overline{2}$, $f(\overline{2}) = \overline{4}$ and $f(\overline{3}) = \overline{3}$. Hence f is a bijection and defines a group isomorphism $(\mathbb{Z}_4, +) \cong (\mathbb{Z}_5^*, \cdot)$.

Definition 4.7. Let G be a group. A *subgroup* H of G is a subset of G, which contains the identity element and is closed under the law of composition and inverse. ◊

This can be combined into one equivalent condition:

Proposition 4.8. *H is a subgroup of a group G if and only if $a\,b^{-1} \in H$ for all $a, b \in H$.*

Example 4.9. (1) Let $G = (\mathbb{R} \setminus \{0\}, \cdot)$. Then the set of strictly positive numbers $H = (\mathbb{R}^+, \cdot)$ forms a subgroup of G since the multiplicative inverse of a positive element is positive and the product of two positive numbers is again positive. The nonzero integers $(\mathbb{Z}\setminus\{0\}, \cdot)$ do not form a subgroup of G, because the inverse elements (with the exception of 1 and -1) are not integers.

(2) Let $G = (\mathbb{Z}, +)$ and $H = 26\,\mathbb{Z}$ be the set of all integer multiples of 26. Then H is an additive subgroup of G. ◊

The following statement is easy to show.

Proposition 4.10. *Let $f : G_1 \to G_2$ be a group homomorphism. The kernel of f, i.e., the set $\{g_1 \in G_1 \mid f(g_1) = e_2\}$, where e_2 is the identity element in G_2, is a subgroup of G_1. Furthermore, the image im(f) is a subgroup of G_2.* ◊

Each group element generates a subgroup:

Definition 4.11. Let G be a group and $g \in G$; then the set $\langle g \rangle = \{g^k \mid k \in \mathbb{Z}\}$ is called the *subgroup generated by* g. Note that we used the multiplicative notation. For an additive group, we would write $\langle g \rangle = \{k \cdot g \mid k \in \mathbb{Z}\}$ instead. ◊

The subgroup $\langle g \rangle$ is in fact a *cyclic* group (see Definition 4.18 below). Next, we define the *order* of a group and the order of elements:

Definition 4.12. Let G be a group; then the *order of the group G* is defined to be the number $|G|$ of elements in G (or infinity) and denoted by $\mathrm{ord}(G)$. Now let $g \in G$. Then the *order of the element* g is defined by the order of the subgroup $\langle g \rangle$ generated by g, i.e., $\mathrm{ord}(g) = \mathrm{ord}(\langle g \rangle)$. $\diamondsuit$

We refer to [**Sho09**] for a proof of the following theorem.

Theorem 4.13. (*Lagrange*) *Suppose G is a finite group and $H \subset G$ is a subgroup. Then the order of H divides the order of G:*

$$\mathrm{ord}(H) \mid \mathrm{ord}(G). \qquad \diamondsuit$$

If $H = \langle g \rangle$, then the Theorem of Lagrange implies:

Corollary 4.14. *Let G be a finite group and $g \in G$; then the group order is divisible by the element order:*

$$\mathrm{ord}(g) \mid \mathrm{ord}(G). \qquad \diamondsuit$$

We obtain Euler's Theorem:

Theorem 4.15. (*Euler's Theorem*) *Let G be a finite group and $g \in G$; then*

$$g^{\mathrm{ord}(G)} = e.$$

Proof. Let n be the smallest positive integer such that $g^n = e$. It follows that $n = \mathrm{ord}(g)$, since $\langle g \rangle$ has exactly $\mathrm{ord}(g)$ elements. Hence

$$(g^n)^k = g^{\mathrm{ord}(g)\,k} = e$$

for any $k \in \mathbb{N}$. Then the assertion follows from Corollary 4.14. $\square$

Euler's Theorem is often stated for $G = \mathbb{Z}_n^*$. Since $\mathrm{ord}(\mathbb{Z}_n^*) = \varphi(n)$ (see Definition 3.18), we obtain

$$x^{\varphi(n)} \equiv 1 \mod n$$

for any $x \in \mathbb{Z}$ with $\gcd(x, n) = 1$. If n is a prime number p, then one has

$$x^{p-1} \equiv 1 \mod p$$

for any integer x which is not a multiple of p. This implies *Fermat's Little Theorem*:

$$x^p \equiv x \mod p.$$

This modular equation holds for any prime number p and integer x.

Euler's Theorem shows how the exponent can be reduced in a modular exponentiation.

Proposition 4.16. *Let $x \in \mathbb{Z}$, $n, a_1, a_2 \in \mathbb{N}$ and assume that $a_1 \equiv a_2 \mod \varphi(n)$. Then*
$$x^{a_1} \equiv x^{a_2} \mod n\,.$$

Proof. By assumption, $a_1 - a_2$ is a multiple of $\varphi(n)$. Then Euler's Theorem implies $x^{a_1 - a_2} \equiv 1 \mod n$. □

Note that the exponent can be reduced modulo $\varphi(n)$, but not modulo n (compare Warning 3.23).

Example 4.17. Calculate $7^{22} \mod 11$. Since $\varphi(11) = 10$ and $22 \equiv 2 \mod 10$, we obtain $7^{22} \equiv 7^2 = 49 \equiv 5 \mod 11$. ◊

Now we will study an important type of group, the *cyclic groups*.

Definition 4.18. Let G be a group and $g \in G$. If $\langle g \rangle = G$, then G is called a *cyclic group* and we say g is a *generator* of G. ◊

The elements of a cyclic group with generator g are
$$\langle g \rangle = \{\dots, g^{-2}, g^{-1}, e, g, g^2, g^3, \dots\}.$$
If a cyclic group has finite order n, then $g^n = e$, $g^{n+1} = g$, $g^{n+2} = g^2$, ... and also $g^{-1} = g^{n-1}, g^{-2} = g^{n-2}, \dots$ (see Figure 4.1). Hence
$$\langle g \rangle = \{e, g, g^2, \dots, g^{n-1}\}.$$

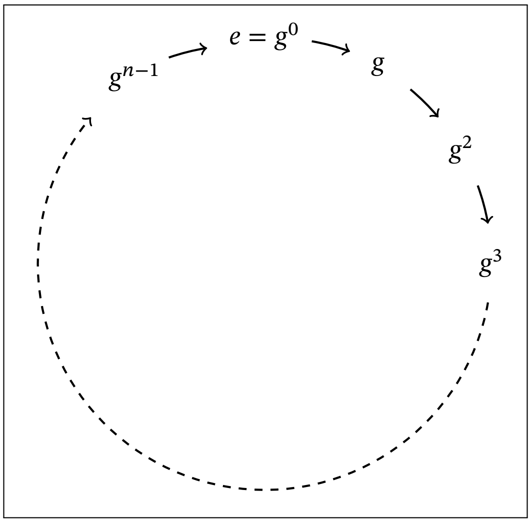


Figure 4.1. Cyclic group of order n generated by g.

Example 4.19. (1) $(\mathbb{Z}_n, +)$ is generated by $\overline{1}$, i.e., $\mathbb{Z}_n = \langle\overline{1}\rangle$. Hence $\mathbb{Z}_n$ is a cyclic group of order n.

(2) $(\mathbb{Z}, +)$ is a cyclic group of infinite order and $\mathbb{Z} = \langle 1\rangle = \langle -1\rangle$. ♢

In fact, one can show that all cyclic groups are isomorphic to either $(\mathbb{Z}_n, +)$ or $(\mathbb{Z}, +)$.

Proposition 4.20. *Let G be a cyclic group. If* $\mathrm{ord}(G) = n$*, then G is isomorphic to $\mathbb{Z}_n$. If* $\mathrm{ord}(G) = \infty$*, then G is isomorphic to $\mathbb{Z}$.*

Proof. Let $G = \langle g\rangle$. We use the multiplicative notation for G. If g has infinite order, then $f : \mathbb{Z} \to G$ with $f(k) = g^k$ defines an isomorphism from the additive group $(\mathbb{Z}, +)$ to G. If $\mathrm{ord}(g) = n$, then $f : \mathbb{Z}_n \to G$, $f(k \mod n) = g^k$ gives an isomorphism. □

Example 4.21. $(\mathbb{Z}_5^*, \cdot)$ is generated by $2 \mod 5$ and is cyclic of order 4. We have seen above (Example 4.6) that $f(k \mod 4) = 2^k \mod 5$ gives an isomorphism between the additive group $\mathbb{Z}_4$ and the multiplicative group $\mathbb{Z}_5^*$. Hence two groups that look different may still be isomorphic. ♢

How can one find or verify generators of a finite cyclic group? If the group is large, then it is inefficient or even computationally impossible to compute the sequence of powers $g, g^2, g^3, \ldots$ and to check whether all elements of G occur. We can use the following observation to give a more efficient method: suppose that $\mathrm{ord}(G) = n$. If g is not a generator, then $\mathrm{ord}(g)$ is strictly less than n and divides $\frac{n}{q}$ for some prime factor q of n. In this case, we have $g^{n/q} = 1$. Hence we only need to check the powers $g^{n/q}$ for all prime factors q of n. If all powers are different from the identity element, then $\mathrm{ord}(g) = n$ and g is a generator.

The following Algorithm 4.1 takes a finite cyclic group G, the group order n and an element $g \in G$ as input and outputs TRUE if g is a generator and otherwise FALSE.

Algorithm 4.1 Generator of a Cyclic Group

```
Input: Cyclic group $G$ of order $n \in \mathbb{N}$, identity element 1 and $g \in G$.
Output: TRUE, if $\mathrm{ord}(g) = n$. Otherwise, return FALSE.
Initialisation: result=TRUE
1: for each prime factor $q$ of $n$ do
2:     $b = g^{n/q}$
3:     if $b = 1$ then
4:         result=FALSE
5:         break
6:     end if
7: end for
8: return result
```

Example 4.22. Let $G = \mathbb{Z}_{53}^*$. We want to check whether $g = 2 \mod 53$ is a generator of G. Since 53 is a prime, we have $\text{ord}(G) = 52$. The factorization $52 = 2^2 \cdot 13$ yields the prime factors 2 and 13. One computes $g^{52/13} = 2^4 = 16 \mod 53$ and $g^{52/2} = 2^{26} \equiv 52 \mod 53$. We conclude that that $g = 2$ is a generator of G. It also follows that $g^2 = 4$ has order 26 and the order of $g^4 = 16$ is 13.

Now consider $h = 7 \mod 53$; then $7^4 \equiv 16 \mod 53$ and $h^{26} \equiv 1 \mod 53$. Hence 7 is not a generator of G. Since $\text{ord}(h)$ divides $\text{ord}(G) = 52$, the order of h must be 1, 2, 13 or 26. Because $h^2 \equiv 49 \mod 53$ and $h^{13} \equiv 52 \mod 53$ are not congruent to 1, we conclude that $\text{ord}(h) = 26$.

Definition 4.23. A generator of the multiplicative group $\mathbb{Z}_n^*$ is called a *primitive root* modulo n. ♢

The following Theorem states that primitive roots modulo prime numbers exist.

Theorem 4.24. *Let p be a prime; then $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$. The number of primitive roots is $\varphi(p-1)$.* ♢

There are several (non-trivial) proofs of this theorem. Note that there are certain composite numbers, for example $n = 12$, such that $\mathbb{Z}_n^*$ does not possess a primitive root.

Example 4.25. Let $p = 253530120045645880299340641 2663$. We use SageMath to compute element orders in $\mathbb{Z}_p^*$ and to find a primitive root. First, we verify that p is prime and factorize $p-1$.

```
sage: p=2535301200456458802993406412663;is_prime(p);factor(p-1)
True
2 * 1267650600228229401496703206331
```

The prime factors of $p-1$ are 2 and $q = 1267650600228229401496703206331$. Hence $\text{ord}(g) \in \{1, 2, q, p-1\}$ for all $g \in \mathbb{Z}_p^*$. Obviously, only $g \equiv 1$ has order 1 and only $g \equiv -1$ has order 2. The other residue classes have either prime order q or maximal order $2q = p-1$. If $g^q \equiv 1 \mod p$, then $\text{ord}(g) = q$. Otherwise, $\text{ord}(q) = p-1$ and g is a primitive root. For example, $\text{ord}(3 \mod p) = q$ and $\text{ord}(7 \mod p) = p-1$.

```
sage: q=1267650600228229401496703206331
      power_mod(3,q,p)
1
sage: power_mod(7,q,p)
2535301200456458802993406412662
```

The *Chinese Remainder Theorem* provides a decomposition of cyclic groups.

Theorem 4.26. (*Chinese Remainder Theorem*) *Let $a, b \in \mathbb{N}$ be relatively prime, i.e.,* $\gcd(a, b) = 1$. *Set $n = ab$. Then the natural map $f : \mathbb{Z}_n \to \mathbb{Z}_a \times \mathbb{Z}_b$, given by*

$f(k \bmod n) = (k \bmod a,\ k \bmod b)$, is well defined and gives an isomorphism of additive groups:

$$\mathbb{Z}_n \cong \mathbb{Z}_a \times \mathbb{Z}_b.$$

The restriction of this map to the group of units yields an isomorphism of multiplicative groups:

$$\mathbb{Z}_n^* \cong \mathbb{Z}_a^* \times \mathbb{Z}_b^*.$$

Proof. We prove the isomorphism of the additive groups. Since a and b divide n, the map is well defined. It follows from the definition of f that the map is a homomorphism. Since $\mathbb{Z}_n$ and $\mathbb{Z}_a \times \mathbb{Z}_b$ both contain $n = ab$ elements, it suffices to prove the surjectivity. Let $(k_1 \bmod a,\ k_2 \bmod b) \in \mathbb{Z}_a \times \mathbb{Z}_b$. We need to find an element $k \in \mathbb{Z}$ such that $k \equiv k_1 \bmod a$ and $k \equiv k_2 \bmod b$. Since $\gcd(a, b) = 1$, the Extended Euclidean Algorithm gives $x, y \in \mathbb{Z}$ such that $ax + by = 1$. This equation implies $ax \equiv 1 \bmod b$ and $by \equiv 1 \bmod a$. Now we set

$$k = k_1 by + k_2 ax.$$

Then $k \equiv k_1 by \equiv k_1 \bmod a$ and $k \equiv k_2 ax \equiv k_2 \bmod b$, as desired. □

If p and q are different prime numbers and $n = pq$, then

$$\mathbb{Z}_n \cong \mathbb{Z}_p \times \mathbb{Z}_q \text{ and } \mathbb{Z}_n^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_q.$$

Furthermore, we obtain $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$. These groups play an important role in the RSA algorithm (see Section 9.2).

Remark 4.27. The Chinese Remainder Theorem also holds true for more than two factors if the factors are pairwise relatively prime.

Example 4.28. Let $n = 60 = 2^2 \cdot 3 \cdot 5$. Then the Chinese Remainder Theorem gives the following decompositions of $\mathbb{Z}_{60}$:

$$\mathbb{Z}_{60} \cong \mathbb{Z}_4 \times \mathbb{Z}_{15} \cong \mathbb{Z}_4 \times \mathbb{Z}_3 \times \mathbb{Z}_5.$$

But note that $\mathbb{Z}_4$ is not isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$. Both groups have order 4, but the first group is cyclic while the second is not. ♢

Cyclic groups form the main building block in the classification of arbitrary finite abelian groups.

Theorem 4.29. (*Fundamental Theorem of Abelian Groups*) *Let G be a finite abelian group. Then G is isomorphic to a direct product of cyclic groups $\mathbb{Z}_{p^k}$ of order p^k, where p is a prime number and $k \in \mathbb{N}$. The same prime p can appear in several factors.*

Proof. According to Proposition 4.20, cyclic groups of order n are isomorphic to $\mathbb{Z}_n$. Suppose

$$n = p_1^{k_1} \cdots p_n^{k_n}$$

is the prime factorization of n. Then the Chinese Remainder Theorem implies

$$\mathbb{Z}_n \cong \mathbb{Z}_{p_1^{k_1}} \times \cdots \times \mathbb{Z}_{p_n^{k_n}}.$$

It remains to show that every finite abelian group is a product of cyclic groups (see for instance [**Sho09**]). □

Example 4.30. (1) Let G be an abelian group of order 77. Then $G \cong \mathbb{Z}_7 \times \mathbb{Z}_{11}$. G is isomorphic to $\mathbb{Z}_{77}$ and is cyclic.

(2) Suppose G is an abelian group of order 18. Then G is either isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_9$ or to $\mathbb{Z}_2 \times \mathbb{Z}_3 \times \mathbb{Z}_3$. Note that these two groups are not isomorphic. The first group is cyclic and the second group is not cyclic.

4.2. Rings and Fields

Rings are sets with *two* operations:

Definition 4.31. A *ring* (or, more precisely, a commutative ring with unity) is a set R with two operations (addition $+$ and multiplication $\cdot$) and the following properties:

- $(R, +)$ is an abelian group with the additive identity element 0.
- $(R, \cdot)$ satisfies the associative law, is commutative and has an identity element denoted by 1. The existence of an inverse element is not required.
- $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ for all $x, y, z \in R$ (distributivity).

Example 4.32. The integers $\mathbb{Z}$ and $\mathbb{Z}_n$, the integers modulo n, form a ring with respect to addition and multiplication of integers and residue classes, respectively. ♢

Maps between rings that are compatible with addition *and* multiplication are called *ring homomorphisms*.

Definition 4.33. Let $f : R_1 \to R_2$ be a map between the rings R_1 and R_2. Then f is called a *ring homomorphism* if

(1) $f(x + y) = f(x) + f(y)$ for all $x, y \in R$, and

(2) $f(x \cdot y) = f(x) \cdot f(y)$ for all $x, y \in R$, and

(3) $f(1) = 1$.

A bijective ring homomorphism is called an *isomorphism*, and one writes $R_1 \cong R_2$.

Example 4.34. Let a, $b \in \mathbb{N}$ be relatively prime and $n = ab$. Then the Chinese Remainder Theorem 4.26 gives a *ring isomorphism*

$$\mathbb{Z}_n \cong \mathbb{Z}_a \times \mathbb{Z}_b.$$

Definition 4.35. Let R be a ring. Then the subset of invertible elements with respect to multiplication is called the *units* of R and is denoted by R^*. The units form an abelian group. ♢

Definition 4.35 generalizes Definition 3.18 where we defined the units $\mathbb{Z}_n^*$ of the integers modulo n.

Example 4.36. $\mathbb{Z}^* = \{-1, 1\}$. This group is isomorphic to the additive group $\mathbb{Z}_2$. ♢

Clearly, the additive identity element $0 \in R$ is not multiplicatively invertible, but if all other elements of R are invertible, then R is called a *field*.

Definition 4.37. A *field* K is a ring where all nonzero elements are units, i.e., $K^* = K \setminus \{0\}$.

Example 4.38. $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ are fields, but $\mathbb{Z}$ is not a field.

Definition 4.39. Let F be a field and $K \subset F$. If K is a field with respect to the field operations inherited from F, then we say that K is a *subfield* of F or F is a *field extension* of K. ♢

It is evident that a field extension F of K is also a vector space over K (see Section 4.4 on vector spaces).

Definition 4.40. Let F be a field extension of K. If the dimension of F over K (as a K-vector space) is finite and equal to n, then n is called the *degree* of the field extension and we write $[F : K] = n$.

Example 4.41. (1) $\mathbb{R}$ is a subfield of $\mathbb{C}$. Furthermore, $\mathbb{C}$ is a vector space over $\mathbb{R}$ and $[\mathbb{C} : \mathbb{R}] = 2$. A basis is given by 1 and i, where $i \in \mathbb{C}$ is the imaginary unit.

(2) $\mathbb{R}$ is a field extension of $\mathbb{Q}$, but the degree is infinite. ♢

The following section deals with *finite fields* and their extensions.

4.3. Finite Fields

We are particularly interested in *finite fields*. A natural candidate is the ring of integers modulo n, but this only gives a field if n is a prime number:

Proposition 4.42. *Let $n \geq 2$. Then $(\mathbb{Z}_n, +, \cdot)$ is a field if and only if n is a prime number.*

Proof. Suppose n is prime. Then the integers $1, 2, \dots, n-1$ are relatively prime to n and hence invertible modulo n (see Proposition 3.17). Conversely, if these numbers are invertible and therefore relatively prime to n, then n has no non-trivial divisors and must be a prime number. □

Warning 4.43. In algebraic number theory, the ring of *p-adic integers* is denoted by $\mathbb{Z}_p$. This infinite ring is the p-adic completion of $\mathbb{Z}$ and not needed in our context. In cryptography, $\mathbb{Z}_n$ stands for the finite cyclic group of order n and $\mathbb{Z}_p$ is the finite cyclic group of prime order p. However, $\mathbb{Z}/(n) = \mathbb{Z}/n\mathbb{Z}$ and $\mathbb{Z}/(p) = \mathbb{Z}/p\mathbb{Z}$ is a clearer notation.

Definition 4.44. Let p be a prime number; then we write $GF(p)$ or $\mathbb{F}_p$ for the *field* $(\mathbb{Z}_p, +, \cdot)$ with p elements.

Example 4.45. The binary digits $\{0, 1\}$ form a field which is isomorphic to $GF(2)$, where addition is XOR and multiplication is AND (see Table 1.1 in Chapter 1). ♢

There are *rings* of any finite order (for example $\mathbb{Z}_n$), but this is not the case for fields:

Proposition 4.46. *Let K be a finite field of order n; then n is a prime number or a prime-power.*

Proof. Let 1 be the multiplicative identity element in K. By assumption, $\operatorname{ord}(K) = n$ and therefore $n \cdot 1 = 0$ in K. At least one prime factor p of n must be zero in K, i.e., $p \cdot 1 = 0$ in K, because otherwise all prime factors are invertible, which is impossible since their product is 0. This implies that $GF(p)$ is a subfield of K. In fact, $GF(p)$ is the image of the ring homomorphism $\mathbb{Z} \to K$ that sends $1 \in \mathbb{Z}$ to $1 \in K$. By Definition 4.39, K is a field extension and thus a vector space over $GF(p)$. Since K is a finite field, the degree of K over $GF(p)$ must be finite, say $[K : GF(p)] = m$. Hence $\operatorname{ord}(K) = p^m$, which proves the assertion. □

Definition 4.47. Let K be a field. The *characteristic* of K is defined to be the smallest positive integer n such that $n \cdot 1 = 0$ in K. If $n \cdot 1 \neq 0$ in K for all nonzero integers n, then K is said to have characteristic 0. The characteristic of a field K is denoted by $char(K)$.

Example 4.48. $char(\mathbb{Q}) = char(\mathbb{R}) = char(\mathbb{C}) = 0$. For a prime number p, $char(GF(p)) = p$. ♢

However, $GF(p)$ is not the only field of characteristic p, and in the following we construct fields of order p^n for primes p and integers $n \geq 2$. Unfortunately, the naive constructions do not work:

- $\mathbb{Z}_{p^n}$ is *a ring* with p^n elements, but *not a field* (compare Proposition 4.42). For example, p is nonzero but not invertible modulo p^n since $\gcd(p, p^n) = p$.
- $\mathbb{Z}_p^n = \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p$ with component-wise addition and multiplication is a ring with p^n elements, but *not a field* (see Exercise 12). ♢

In fact, the construction of a field $GF(p^n)$ of order p^n is a bit more involved and requires polynomial rings.

Definition 4.49. Let K be a field. Then $K[x]$ is called the *set (or ring) of polynomials* over K and consists of all formal expressions

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where $a_i \in K$ and $n \geq 0$ is an integer. The *degree* $\deg(f)$ of f is equal to n if $a_n \neq 0$. The degree of constant polynomials is 0. A polynomial is called *monic* if $a_n = 1$. ◊

Polynomials can be added and multiplied in the obvious way.

Proposition 4.50. *The polynomials* $(K[x], +, \cdot)$ *over K form a ring.*

Example 4.51. Let $f, g \in GF(2)[x]$ with $f(x) = 1 + x + x^3$ and $g(x) = x + x^2$. We compute

$$\begin{aligned} f(x) + g(x) &= (1 + x + x^3) + (x + x^2) = 1 + x^2 + x^3, \\ f(x) \cdot g(x) &= (1 + x + x^3)(x + x^2) \\ &= x + x^2 + x^2 + x^3 + x^4 + x^5 = x + x^3 + x^4 + x^5. \end{aligned}$$ ◊

Note that $K[x]$ *is not a field*, since polynomials of degree ≥ 1 cannot be inverted multiplicatively.

But we have a *division with remainder*. Let $f(x)$, $g(x) \in K[x]$ with $g(x) \neq 0$. Then the division $f(x) : g(x)$ gives a quotient $q(x) \in K[x]$ and a remainder $r(x) \in K[x]$ such that

$$f(x) = q(x)g(x) + r(x) \text{ and } \deg(r) < \deg(g).$$

Obviously, $g(x)$ divides $f(x)$ if and only if the remainder is 0.

Example 4.52. Let $f(x) = x^6 + x^5 + x^3 + x^2 + x + 1$ and $g(x) = x^4 + x^3 + 1$ be polynomials in $GF(2)[x]$. The quotient of $f(x) : g(x)$ is $q(x) = x^2$, the remainder is $r(x) = x^3 + x + 1$ and we have an equation

$$x^6 + x^5 + x^3 + x^2 + x + 1 = x^2(x^4 + x^3 + 1) + (x^3 + x + 1).$$

SageMath can do these calculations:

```
sage: R.<x> = PolynomialRing(GF(2),x)
sage: f = x^6+x^5+x^3+x^2+x+1
sage: g = x^4+x^3+1
sage: q= f // g
sage: r= f % g
sage: print q, r, q*g+r
x^2     x^3 + x + 1     x^6 + x^5 + x^3 + x^2 + x + 1
```

Definition 4.53. Let $f(x), g(x) \in K[x]$ be nonzero polynomials. Then the *greatest common divisor* $\gcd(f, g)$ is the monic polynomial of highest possible degree that divides $f(x)$ and $g(x)$. ◊

The greatest common divisor (gcd) of polynomials can be efficiently computed using the *Extended Euclidean Algorithm*, analogous to the gcd of integers (see Algorithm 3.1 in Chapter 3). The integer division is replaced by the division of polynomials with remainder. The Extended Euclidean Algorithm takes two polynomials f and g as input and outputs $gcd(f, g)$ along with two polynomials $a(x)$ and $b(x)$ such that

$$gcd(f, g) = a(x)f(x) + b(x)g(x).$$

The gcd of f and g has the following property: if $h(x)$ divides $f(x)$ and $g(x)$, then $h(x)$ divides $\gcd(f, g)$.

Example 4.54. We compute $\gcd(x^3+1, x^2+1)$ over $GF(2)$ (see Table 4.1) and obtain

$$\gcd(x^3+1, x^2+1) = x+1 = 1\cdot(x^3+1) - x\cdot(x^2+1).$$

Table 4.1. Computation of $\gcd(x^3+1, x^2+1) = x+1$.

$(x^3+1) : (x^2+1) = x$ rem. $x+1$	$(x^3+1) = x\cdot(x^2+1) + (x+1)$
$(x^2+1) : (x+1) = (x+1)$ rem. 0	$(x^2+1) = (x+1)\cdot(x+1) + 0$

There is a *formal derivative* of polynomials:

Definition 4.55. The *formal derivative* is a map $D : K[x] \to K[x]$ defined by $D(x^n) = n\,x^{n-1}$, and a K-linear extension to arbitrary polynomials:

$$D(a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0) = na_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \cdots + a_1.$$

Example 4.56. Let $n \in \mathbb{N}$ and $f(x) = x^{2^n} + x \in GF(2)[x]$. Then

$$D(f) = 2^n x^{2^n-1} + 1 = 1,$$

since $2^n = 0$ in $GF(2)$. ◊

One can show that the derivative satisfies the product rule (see Exercise 13):

$$D(f\cdot g) = f\cdot D(g) + D(f)\cdot g.$$

Note that D does not have a geometric interpretation as in the real case. However, the derivative can detect double roots of polynomials.

Proposition 4.57. *Let $f(x) \in K[x]$ and assume that* $\gcd(f, D(f)) = 1$. *Then $f(x)$ is square-free, i.e., it is not divisible by the square of any polynomial of degree at least* 1. *In particular, $f(x)$ is not divisible by $(x-a)^2$ for any $a \in K$ and does not have double roots.*

Proof. Suppose that f is not square-free and $f = g^2 h$ for polynomials $g,\ h \in K[x]$ with $\deg(g) \geq 1$; then

$$D(f) = D(g^2)h + g^2 D(h) = 2gD(g)h + g^2 D(h).$$

Hence g is a common factor of f and $D(f)$, a contradiction to $\gcd(f, D(f)) = 1$. □

Example 4.58. Let $n \in \mathbb{N}$ and $f(x) = x^{2^n} + x \in GF(2)[x]$ (see above). Since $\gcd(f, D(f)) = 1$, the polynomial f does not have multiple roots. ◊

Now we define *residue classes* of polynomial rings:

Definition 4.59. Let $g \in K[x]$ be a polynomial with $\deg(g) \geq 1$. Then $g(x)$ defines an equivalence relation on $K[x]$:

$$f_1(x) \sim f_2(x) \text{ if } f_1(x) - f_2(x) = q(x)g(x) \text{ for some } q(x) \in K[x].$$

Equivalent polynomials f_1 and f_2 are called *congruent modulo* $g(x)$ and we write $f_1(x) \equiv f_2(x) \mod g(x)$. The set of equivalence classes or *residue classes modulo* $g(x)$ is denoted by $K[x]/(g(x))$. ♢

Two polynomials f_1 and f_2 are congruent modulo g if and only if they have the same remainder when divided by $g(x)$. Note that the definition is similar to residue classes modulo an integer n, but here the construction is based on the polynomial ring $K[x]$ instead of the ring of integers $\mathbb{Z}$.

The residue classes modulo $g(x)$ form not only a set, but also a ring:

Proposition 4.60. *Let $g \in K[x]$ and $n = \deg(g) \geq 1$. Then $K[x]/(g(x))$ is again a ring called a quotient ring, factor ring or residue class ring, with the operations induced by $K[x]$. Each residue class has a unique standard representative of degree less than n.*

Proof. The ring structure can be easily verified. The standard representative can be found by division with remainder. Let $f(x) \in K[x]$ be any representative of a residue class. We divide $f(x)$ by $g(x)$ and obtain polynomials $q(x)$, $r(x)$ such that

$$f(x) = q(x)g(x) + r(x),$$

where $\deg(r) < n$. The equation implies $f(x) \equiv r(x) \mod g(x)$, where $r(x)$ is the standard representative. □

Example 4.61. We continue Example 4.52. The division with remainder implies

$$x^6 + x^5 + x^3 + x^2 + x + 1 \equiv x^3 + x + 1 \mod x^4 + x^3 + 1.$$

Therefore, the classes of $x^6 + x^5 + x^3 + x^2 + x + 1$ and $x^3 + x + 1$ are equal in the residue class ring $GF(2)[x]/(x^4 + x^3 + 1)$.

Remark 4.62. The construction of residue classes can be studied in a more general context. Let R be a ring. An *ideal* $I \subset R$ is an additive subgroup with the property that $r \cdot x \in I$ for all $r \in R$ and $x \in I$. For any ideal I of a ring R one has the *quotient ring* R/I. Two elements $x, y \in R$ are equivalent and identified in R/I if $x - y \in I$.

We considered polynomial rings $R = K[x]$ and *principal ideals* $I = (g(x))$ generated by a single polynomial $g(x) \in K[x]$. If $R = \mathbb{Z}$ and $I = (n)$, then the quotient R/I defines the integers modulo n, i.e., $\mathbb{Z}/(n) = \mathbb{Z}_n$.

Proposition 4.63. *Let p be a prime and $g \in GF(p)[x]$ a polynomial of degree n; then the quotient ring $GF(p)[x]/(g(x))$ has p^n elements.*

Proof. The elements of $GF(p)[x]/(g(x))$ are in bijection with the polynomials in $GF(p)[x]$ of degree less than n (see Proposition 4.60). Such polynomials are given by n coefficients in $GF(p)$. Hence the number of elements in the quotient ring is p^n. □

The polynomial ring $K[x]$ has similar properties to $\mathbb{Z}$ with respect to factorization. Polynomials can be decomposed into a product of polynomials and the factorization is essentially unique. The prime numbers in $\mathbb{Z}$ correspond to *irreducible* polynomials in $K[x]$.

Definition 4.64. A polynomial $f(x) \in K[x]$ is called *irreducible*, if it cannot be factored into two polynomials of smaller degree. Otherwise, the polynomial is called *reducible*. ♢

Irreducible polynomials in $K[x]$ do not possess any zeros $a \in K$, since otherwise a linear factor $(x - a)$ can be split off. The converse statement only holds for degree ≤ 3. There are polynomials of degree ≥ 4 without zeros which are reducible (see Example 4.65 (4) below).

Example 4.65. (1) $f(x) = x^2 + 1$ is irreducible in $\mathbb{R}[x]$, but reducible in $\mathbb{C}[x]$:

$$x^2 + 1 = (x - i)(x + i).$$

$f(x)$ is also reducible over $GF(2)$, since $x^2 + 1 = (1 + x)^2$ in $GF(2)[x]$.

(2) $f(x) = x^2 + x + 1$ is irreducible in $GF(2)[x]$. Assume that $f(x)$ is reducible. Then $f(x)$ factorizes into two linear factors of type $(x - a)$, which is impossible since $f(x)$ does not have a root in $GF(2)$ (note that $f(0) = 1$ and $f(1) = 1$).

(3) $f(x) = x^2 + 1$ is irreducible over $GF(3)[x]$, since $f(0)$, $f(1)$ and $f(2)$ are nonzero modulo 3.

(4) $g(x) = x^4 + x^2 + 1$ has no zeros over $GF(2)$, but $g(x) = (x^2 + x + 1)^2$ in $GF(2)[x]$. Hence $g(x)$ is reducible.

Table 4.2 lists the irreducible polynomials over $GF(2)$ of degree up to 5.

Example 4.66. Let $g(x) = x^8 + x^4 + x^3 + x + 1 \in GF(2)[x]$. This polynomial is used by the AES block cipher (see Section 5.2). We use SageMath to verify that $g(x)$ is irreducible:

```
sage: R.<x> = PolynomialRing(GF(2),x)
sage: g=x^8+x^4+x^3+x+1
sage: g.is_irreducible()
True
```

An important fact is that polynomial rings modulo irreducible polynomials are fields:

Proposition 4.67. *Let $g(x) \in K[x]$ be an irreducible polynomial. Then the quotient ring $K[x]/(g(x))$ is a field.* ♢

Table 4.2. Irreducible polynomials over $GF(2)$.

Degree	Polynomials
2	$x^2 + x + 1$
3	$x^3 + x + 1$
	$x^3 + x^2 + 1$
4	$x^4 + x + 1$
	$x^4 + x^3 + x^2 + x + 1$
	$x^4 + x^3 + 1$
5	$x^5 + x^2 + 1$
	$x^5 + x^3 + x^2 + x + 1$
	$x^5 + x^3 + 1$
	$x^5 + x^4 + x^3 + x + 1$
	$x^5 + x^4 + x^3 + x^2 + 1$
	$x^5 + x^4 + x^2 + x + 1$

This result might look surprising, since $K[x]$ is far from being a field: no polynomial of degree ≥ 1 is multiplicatively invertible in $K[x]$. However, an inversion modulo $g(x)$ is often possible, since two representatives f_1 and f_2 are multiplicative inverses mod $g(x)$ if

$$f_1(x)f_2(x) = 1 + q(x)g(x)$$

for some $q(x) \in K[x]$.

The proof of Proposition 4.67 uses the *Extended Euclidean Algorithm for polynomials*. We briefly sketch the proof: let $f(x)$ be a nonzero polynomial of degree less than $\deg(g)$. Then there are polynomials h_1 and h_2 such that

$$1 = h_1(x)f(x) + h_2(x)g(x).$$

This shows that $1 \equiv h_1(x)f(x) \mod g(x)$ so that $f(x)$ is invertible modulo $g(x)$.

Definition 4.68. Let $g(x) \in GF(p)[x]$ be an *irreducible polynomial* of degree n. Then the residue field $GF(p)[x]/(g(x))$ defines the *Galois Field* $GF(p^n) = \mathbb{F}_{p^n}$ of order p^n. ♢

It follows from Proposition 4.63 that the field $GF(p^n)$ indeed contains p^n elements; each residue class has a unique representative of degree less than n, i.e., each class is represented by a polynomial $f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ with $a_i \in GF(p)$.

Note that the definition of $GF(p^n)$ depends on an irreducible polynomial of degree n and it is not clear whether such a polynomial exists. We want to show that a finite field of order p^n *exists* and is essentially *unique*.

If $GF(p^n)$ exists, then $\text{ord}(GF(p^n)^*) = p^n - 1$ and thus

$$a^{p^n-1} = 1$$

for all $a \in GF(p^n)^*$. This implies

$$a^{p^n} = a$$

for all $a \in GF(p^n)$, i.e., including the zero element. In other words, the elements of $GF(p^n)$ are exactly the roots of the polynomial

$$f(x) = x^{p^n} - x.$$

Let $a_1, a_2, \dots, a_{p^n}$ denote the elements of $GF(p^n)$. The polynomial $f(x)$ splits into linear factors in $GF(p^n)[x]$, i.e.,

$$x^{p^n} - x = (x - a_1) \cdot (x - a_2) \cdots (x - a_{p^n}).$$

This observation can be used to construct $GF(p^n)$. A general theorem in field theory states that a field can be extended such that any given polynomial of degree ≥ 1 completely factorizes over the extension field.

Theorem 4.69. *Let K be a field and $f(x) \in K[x]$ a polynomial of positive degree. There exists an extension field $F \supset K$ such that $f(x)$ splits into linear factors over F. The smallest field with this property is called the splitting field of f over K. Any two splitting fields are isomorphic.*

Remark 4.70. We refer to the literature (for example [**Sho09**]) for a proof of the above Theorem. It is tempting to construct F as the quotient field of $K[x]$ modulo $f(x)$, but this does not work in general. Firstly, splitting fields exist for *any polynomial* and not only for irreducible ones. Secondly, even if $f(x)$ is irreducible so that $F = K[x]/(f(x))$ gives an extension field of K, the field F might not contain *all roots* of $f(x)$. In this case, further extensions of F are necessary in order to obtain the splitting field.

Example 4.71. (1) The field $\mathbb{C}$ of complex numbers is the splitting field of $f(x) = x^2 + 1$ over $\mathbb{R}$.

(2) Let $K = \mathbb{Q}$ and $f(x) = x^3 - 2$. The polynomial is irreducible over $\mathbb{Q}$ and $F = \mathbb{Q}[x]/(x^3 - 2) = \mathbb{Q}(\sqrt[3]{2})$ is an extension field of degree 3 over $\mathbb{Q}$, but F is not the splitting field of $f(x)$. In fact, F is a subfield of $\mathbb{R}$ and $f(x)$ has two complex roots beside the real root $\sqrt[3]{2}$. One can show that the degree of the splitting field of $f(x)$ over $\mathbb{Q}$ is 6.

Remark 4.72. The *algebraic closure* $\overline{K}$ of a field K has the property that all polynomials completely factor over $\overline{K}$. One can show that an algebraic closure of a field exists and is unique up to isomorphism. For example, $\mathbb{C}$ is the algebraic closure of $\mathbb{R}$. In this case, an extension of degree 2 suffices, but in general, the algebraic closure of a field has infinite degree. This is, for example, true for the fields $\mathbb{Q}$ and $GF(p)$. We note that there are also *transcendent extensions* which are not algebraic, i.e., not defined by adjoining roots of polynomials, for example the field $\mathbb{Q}(x)$ of rational functions in one variable over $\mathbb{Q}$. ◊

We return to the construction of a finite field of order p^n. At this point we know that if $GF(p^n)$ exists, then it is isomorphic to the splitting field of $f(x) = x^{p^n} - x$ over

$GF(p)$. Now we show that the splitting field of f has p^n elements, which proves that $GF(p^n)$ exists (for all $n \in \mathbb{N}$) and is unique up to isomorphism.

Proposition 4.73. *Let $f(x) = x^{p^n} - x \in GF(p)[x]$. The splitting field of $f(x)$ over $GF(p)$ has p^n elements and defines the field $GF(p^n)$.*

Proof. Firstly, we show that $f(x)$ does not have multiple roots. The formal derivative is $D(f) = p^n x^{p^n-1} - 1 = -1$ and thus $\gcd(f, D(f)) = 1$. By Proposition 4.57, f does not have multiple roots and so the splitting field F of $f(x)$ over $GF(p)$ contains at least the p^n distinct roots of $x^{p^n} - x$. However, F could contain more elements. Let $S = \{a_1, \dots, a_{p^n}\}$ be the set of roots. Note that $GF(p) \subset S$ since $a^p \equiv a \bmod p$ for all $a \in GF(p)$.

Next, we show that S forms a *field* which must be equal to F, since F is the smallest field extension of $GF(p)$ where $f(x)$ splits into linear factors. We thus need to prove that $a - b \in S$ for all $a, b \in S$ and $ab^{-1} \in S$ for all $a, b \in S \setminus \{0\}$ (see Proposition 4.8 on conditions for a subgroup). We show that $f(a-b) = 0$ and $f(ab^{-1}) = 0$ if $f(a) = 0$ and $f(b) = 0$. To this end, we observe that

$$(a-b)^{p^m} = a^{p^n} + (-b)^{p^n} = a - b$$

since the other terms given by the Binomial Theorem are multiples of p and therefore zero in $GF(p)$. We note that $(-1)^{p^n} = -1$ if $p \neq 2$ and $-1 = 1$ for $p = 2$. This gives $f(a-b) = 0$. Furthermore,

$$\left(a(b^{-1})\right)^{p^n} = a^{p^n}(b^{-1})^{p^n} = a^{p^n}(b^{p^n})^{-1} = ab^{-1},$$

and so $f(ab^{-1}) = 0$. Summarizing, the roots of $f(x) = x^{p^n} - x$ form the splitting field $F = GF(p^n)$, and this field of p^n elements is unique up to isomorphism. □

Example 4.74. $GF(4)$ is the splitting field of $f(x) = x^4 - x$ over $GF(2)$. This polynomial factorizes into $x(x+1)(x^2+x+1)$ over $GF(2)$. The first two factors correspond to the elements 0 and 1 of the base field $GF(2)$. The polynomial x^2+x+1 is irreducible over $GF(2)$ and

$$GF(4) = GF(2)[x]/(x^2+x+1).$$

$GF(4)$ is represented by the polynomials $\{0, 1, x, x+1\}$.

Addition is obvious (modulo 2), and multiplication also follows the usual rules, but the result is reduced modulo x^2+x+1. For example:

$$x(x+1) = x^2 + x \equiv 1 \mod (x^2+x+1).$$

Table 4.3 shows addition and multiplication in $GF(4)$.

The multiplicative inverses are $1^{-1} = 1$, $x^{-1} \equiv x+1 \mod (x^2+x+1)$ and $(x+1)^{-1} \equiv x \mod (x^2+x+1)$.

Proposition 4.75. *Let n, $m \in \mathbb{N}$. Then $GF(p^m) \subset GF(p^n)$ if and only if $m \mid n$ (see Figure 4.2).*

Table 4.3. Addition and multiplication tables for $GF(4)$.

$+$	0	1	x	$x+1$
0	0	1	x	$x+1$
1	1	0	$x+1$	x
x	x	$x+1$	0	1
$x+1$	$x+1$	x	1	0

$\cdot$	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

Proof. Suppose that $GF(p^m) \subset GF(p^n)$ and the degree of the field extension is $[GF(p^n) : GF(p^m)] = k$. It follows that the order of $GF(p^n)$ is $p^n = (p^m)^k = p^{mk}$ and thus $m \mid n$. To prove the opposite direction, let $a \in GF(p^m)$ and $n = mk$ for some $k \in \mathbb{N}$. Then

$$a^{p^n} = a^{(p^m)^k} = (((a^{p^m})^{p^m})\dots)^{p^m} \text{ (k-fold exponentiation).}$$

Since $a^{p^m} = a$ in each step, we obtain $a^{p^n} = a$ and therefore $a \in GF(p^n)$. $\square$

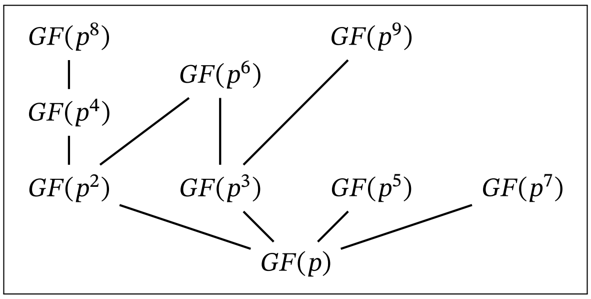


Figure 4.2. Subfield relation for several extension fields of $GF(p)$.

Remark 4.76. As an additive group, $GF(p^n)$ is isomorphic to

$$(\mathbb{Z}_p)^n = \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p.$$

However, the multiplication in $GF(p^n)$ is not defined component-wise (compare Exercise 12), but rather by a multiplication of polynomials. Fixing an irreducible polynomial of degree n yields a unique representation of elements in $GF(p^n)$ as n-dimensional vectors over $GF(p)$.

Example 4.77. Let $g(x) = x^8 + x^4 + x^3 + x + 1 \in GF(2)[x]$ (see Example 4.66). Then $GF(2)[x]/(g(x))$ defines the field $GF(2^8)$ with 256 elements. By fixing $g(x)$, the elements in $GF(2^8)$ are in bijection to polynomials of degree less than 8, which in turn correspond the 8-bit words. The first bit (most significant bit, MSB) corresponds to the coefficient of x^7, the second bit to x^6, etc., and the last bit (least significant bit, LSB) to $x^0 = 1$. The block cipher AES (see Section 5.2) uses this representation of elements in $GF(256)$ as 8-bit words. The elements of this field can also be written in hexadecimal notation. Addition on the 8-bit words is given by a simple XOR operation, but the multiplication is less obvious and defined by a multiplication of polynomials, followed by a reduction modulo $g(x)$.

We can use SageMath for computations in $GF(256)$. Suppose we want to compute $x^7(x+1) \bmod g(x)$ and the multiplicative inverse of $x+1 \bmod g(x)$.

```
sage: R.<x> = PolynomialRing(GF(2),x)
sage: g=x^8+x^4+x^3+x+1
sage: K.<a>=R.quotient_ring(g)
sage: a^7 * (a+1) ; 1/(a+1)
a^7 + a^4 + a^3 + a + 1
a^7 + a^6 + a^5 + a^4 + a^2 + a
```

In hexadecimal notation, the above computations can be written as `80` $\cdot$ `03` $=$ `9B` and `03`$^{-1}$ $=$ `F6`.

The SageMath variable `a` represents the residue class of $x \bmod g(x)$. We could also define the quotient ring by `K=R.quotient_ring(g)`, i.e., without reference to a new variable `a`. In this case, SageMath writes `xbar` for the residue class of x.

4.4. Linear and Affine Maps

Vector spaces, *linear maps* and *affine maps* over finite fields play an important role in cryptography, and we recapitulate some facts from linear algebra.

Definition 4.78. Let K be a field. Then a K-vector space V is an abelian group together with a scalar multiplication, which assigns a vector $\lambda \cdot v \in V$ to a tuple $(\lambda, v) \in K \times V$ such that the following rules are satisfied:

(1) $\lambda \cdot (v+w) = \lambda \cdot v + \lambda \cdot w$ for all $\lambda \in K$, $v, w \in V$.

(2) $(\lambda + \mu) \cdot v = \lambda \cdot v + \mu \cdot v$ for all $\lambda, \mu \in K$, $v \in V$.

(3) $\lambda \cdot (\mu \cdot v) = (\lambda\mu) \cdot v$ for all $\lambda, \mu \in K$, $v \in V$.

(4) $1 \cdot v = v$ for all $v \in V$. ◊

In the following, we assume that the reader knows the definitions of *linear independence*, *basis* and *dimension* of a vector space (see [**WJW⁺14**] or any other textbook on linear algebra).

Example 4.79. (1) Let K be any field; then K^n is the standard example of a n-dimensional K-vector space.

(2) The binary strings of length $n \in \mathbb{N}$ form the $GF(2)$-vector space $GF(2)^n$. The group operation is defined by bitwise XORing. The scalar multiplication is trivial, since the only factors are 0 and 1. Note that there is no natural ring structure on V.

(3) Let p be a prime number and $n \in \mathbb{N}$. Then $GF(p^n)$ is a vector space over $GF(p)$.

(4) $GF(256)^n$ is a $GF(256)$-vector space of dimension n. Here the scalar multiplication is defined over the field $GF(256)$. It is also a $GF(2)$-vector space, but of dimension $8n$.

Maps between vector spaces that are compatible with addition and scalar multiplication are called *linear*:

Definition 4.80. Let $f : V \to W$ be a map between two K-vector spaces. Then f is a K-linear map if:

(1) $f(v_1 + v_2) = f(v_1) + f(v_2)$ for all $v_1, v_2 \in V$ and
(2) $f(\lambda \cdot v) = \lambda \cdot f(v)$ for all $\lambda \in K, v \in V$.

Example 4.81. Let V and W be $GF(2)$-vector spaces; then $f : V \to W$ is linear if $f(v_1 + v_2) = f(v_1) + f(v_2)$ for all $v_1, v_2 \in V$. The second condition is automatically satisfied since the only scalars are 0 and 1.

Remark 4.82. One should understand that linearity is a very strong requirement, and random maps are mostly far from linear! However, linear maps play an important role in many applications as well as in cryptography.

Matrices are a key tool for the description of linear maps. We recapitulate the following fact from linear algebra:

Proposition 4.83. *There is a one-to-one correspondence between $n \times m$ matrices over K and linear maps $f : K^m \to K^n$. Any matrix A over K with n rows and m columns gives a linear map $f : K^m \to K^n$ by setting $f(v) = Av$, where we view v as a column vector.*

Conversely, a linear map $f : K^m \to K^n$ defines a matrix by writing the images of the standard basis, i.e., $f(e_1)$, $f(e_2)$, …, $f(e_m)$, into the columns of a $n \times m$ matrix:

$$A = \begin{pmatrix} | & | & & | \\ f(e_1) & f(e_2) & \dots & f(e_m) \\ | & | & & | \end{pmatrix}. \qquad \lozenge$$

The above construction can be generalized from the standard basis to an arbitrary basis. In fact, a linear map is completely determined by its values on a basis.

Definition 4.84. A K-linear map $f : V \to W$ is said to be an *isomorphism* if f is invertible, i.e., if there is an inverse K-linear map $f^{-1} : W \to V$. $\lozenge$

An isomorphism f between K^m and K^n implies $m = n$, but there are linear maps $f : K^n \to K^n$ which are neither injective nor surjective. f is an isomorphism if and only if the corresponding $n \times n$ matrix A is invertible, i.e., if an inverse $n \times n$ matrix A^{-1} exists. There are several equivalent conditions for A to be invertible, for example that the *determinant* $\det(A)$ is nonzero in K.

Example 4.85. Let $f : GF(2)^3 \to GF(2)^3$ be defined by

$$f(x_1, x_2, x_3) = (x_1 + x_3,\ x_1 + x_2,\ x_2 + x_3).$$

f is linear and the corresponding matrix is

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

The determinant is $\det(A) = 2 \equiv 0 \mod 2$. Hence f is not invertible over $GF(2)$. On the other hand, the corresponding map over the real numbers $\mathbb{R}$ is invertible. $\diamondsuit$

Cryptography primarily considers maps over finite fields, but more recent advances (lattice-based cryptography and quantum computing) also require real and complex vector spaces.

Definition 4.86. An $n \times n$ matrix A over $\mathbb{R}$ is called *orthogonal* if $A^T A = I_n$. $\diamondsuit$

Here A^T denotes the transpose matrix and I_n is the $n \times n$ identity matrix. Orthogonal matrices are invertible, the inverse matrix is $A^{-1} = A^T$ and $\det(A)$ is either 1 or -1. The rows and the columns are *orthonormal* vectors. The associated linear map $f(x) = A\,x$ of real vector spaces is also called orthogonal and preserves lengths and angles.

Example 4.87. The rotation of two-dimensional vectors by α around the origin is described by the following orthogonal matrix:

$$A = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}.$$

One easily verifies that $A^T A = I_2$. The columns of A are obtained by rotating the standard unit vectors e_1 and e_2 by α counter-clockwise around the origin. $\diamondsuit$

The corresponding notion for complex matrices is *unitary*.

Definition 4.88. A complex $n \times n$ matrix A over $\mathbb{C}$ is said to be *unitary* if $A^* A = I_n$, where A^* denotes the conjugate transpose of A, i.e., $A^* = \overline{A}^T$. $\diamondsuit$

The inverse of a unitary matrix A is A^* and $|\det(A)| = |\det(A^*)| = 1$. The rows and columns of A form an orthonormal basis of $\mathbb{C}^n$ with respect to the Hermitian inner product. The associated linear map over $\mathbb{C}$ is also called unitary.

A slight generalization of linear maps are *affine maps*. They differ from linear maps only by a *constant translation*. We remark that affine maps are sometimes also called linear, although this is not fully precise. Furthermore, *nonlinear* usually means that a map is neither linear nor affine.

Definition 4.89. A map $f : V \to W$ between K-vector spaces V and W is called *affine* if there exists a linear map $l : V \to W$ and a vector $b \in W$ such that for all $v \in V$

$$f(v) = l(v) + b. \qquad \Diamond$$

Affine maps $f : K^m \to K^n$ can be described by an $n \times m$ matrix A and a vector $b \in K^n$:

$$f(v) = A\,v + b.$$

Affine maps from $GF(2)^m$ to $GF(2)^n$ correspond to (m, n)-vectorial Boolean maps of algebraic degree 1 (see Definition 1.23).

Example 4.90. Let $f(x_1, x_2, x_3, x_4) = (x_1 + x_2 + x_3 + 1,\ x_3 + x_4)$ be a map over $GF(2)$. Then

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix},\ b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

describes the affine map f, since $f(x) = Ax + b$ for all $x \in GF(2)^4$. $\Diamond$

More examples of linear, affine and nonlinear Boolean functions are given in Example 1.24.

Linear and affine maps play an important role in cryptography. This has several reasons:

- They can be efficiently described by a matrices and vectors, even for large dimensions.
- Matrix computations are efficient, and the running time is polynomial in the number of rows and columns.
- The kernel and the image of a linear map as well as the preimage of any element can be efficiently computed by Gaussian elimination. Also, it can be easily verified whether a linear or affine map is bijective. If an inverse map exists, then it can be efficiently computed and the inverse map is also linear or affine.
- Linear and affine maps over $GF(2)$ can produce *diffusion* and the so-called *avalanche* effect: changing one input bit changes many output bits, if the matrix is appropriately chosen. In fact, flipping the k-th input bit adds the k-th column vector $A\,e_k$ to the output.

However, linear maps also have a major drawback when used in cryptography: there are efficient attacks against encryption schemes that are solely based on linear or affine operations. They do not protect against chosen plaintext attacks. For this reason, linear and *nonlinear operations* are combined in the construction of secure ciphers.

Proposition 4.91. *Let $f : K^m \to K^n$ be an affine map. Suppose the parameters of f (i.e., the corresponding matrix and possibly a translation vector) are secret, but an adversary knows $m+1$ input vectors $v_0, v_1, \dots, v_m$ and the corresponding output vectors $w_i = f(v_i)$,* *where $i = 0, 1, \dots, m$. If $v_1 - v_0, v_2 - v_0, \dots, v_m - v_0$ are linearly independent, then the adversary can efficiently compute the matrix A and the translation vector b such that $f(v) = Av + b$ holds for all $v \in K^m$.*

Proof. Let $f(v) = Av + b$, where A and b are unknown. Since $Av_i + b = w_i$, one has

$$A(v_i - v_0) = w_i - w_0$$

for all $i = 1, 2, \dots, m$. Now write the vectors $v_i - v_0$ and $w_i - w_0$ into the columns of matrices V and W, respectively. The $m \times m$ matrix V is regular since we assumed that the vectors $v_i - v_0$ are linearly independent. Hence

$$AV = W \implies A = WV^{-1},$$

and this provides an efficient matrix formula for A. It remains to compute the translation vector b; to this end we use the equation $b = w_0 - Av_0$. This proves the assertion. □

Example 4.92. Let $E_k : \{0,1\}^{128} \to \{0,1\}^{128}$ be the ciphering function of a block cipher E with block length 128. We identify $\{0,1\}$ with $GF(2)$ and suppose that E_k is affine. Then Proposition 4.91 shows that an adversary, who does not know k, can find the matrix, the translation vector and hence E_k and E_k^{-1} with only 129 known plaintext/ciphertext pairs, if the above independency condition is satisfied. Hence $128 \cdot 129 = 15{,}512$ plaintext/ciphertext bits are sufficient, or slightly more if the vectors are linearly dependent. Note that we made no assumptions concerning k and how the key determined the matrix and the translation vector. This could even be a nonlinear relationship. In fact, the key is not computed during this attack.

Proposition 4.93. *Let F be a keyed family of functions and suppose that all maps F_k are affine; then F is not a pseudorandom function. Similarly, if E is a keyed family of permutations and all maps E_k are affine, then E is not a pseudorandom permutation.*

Proof. Proposition 4.91 shows how an adversary can explicitly compute the parameters of an affine map, i.e., the matrix and the translation vector, using a number of known input/output values. In the distinguishability experiments (see Definitions 2.38 and 2.41), an adversary can then predict $f(m)$ for any input m and compare the result with the response c they obtain from the challenger. If they coincide, then the function f is probably affine and the adversary outputs $b' = 1$. Otherwise, f is random and the adversary outputs $b' = 0$.

Alternatively, an adversary can test whether f is affine, by choosing input values m_1 and m_2 and asking for $f(0), f(m_1), f(m_2), f(m_1 + m_2)$. If f is affine, then

$$f(m_1 + m_2) + f(0) = f(m_1) + f(m_2).$$

An adversary outputs $b' = 1$ if this equation is satisfied, and 0 otherwise. Their advantage is close to 1, and so affine functions cannot be pseudorandom. □

Remark 4.94. The above attack would essentially still work if F_k can be *approximated* by a linear or affine map f, i.e., if F_k and f coincide significantly more often than by chance. Therefore, pseudorandom functions and permutations must be *highly nonlinear*.

4.5. Summary

- Finite cyclic groups of order n are isomorphic to the additive group of integers modulo n.
- Finite abelian groups can be decomposed into a product of cyclic groups of prime-power order.
- The integers modulo a prime number p define the field $GF(p)$.
- The polynomials over a field K form the ring $K[x]$.
- The field $GF(p^n)$ with p^n elements is an extension field of $GF(p)$. It can be defined as the quotient of the polynomial ring over $GF(p)$ modulo an irreducible polynomial of degree n.
- $GF(p^n)$ is the splitting field of the polynomial $x^{p^n} - x$ over $GF(p)$.
- Linear maps between finite-dimensional vector spaces over an arbitrary field can be described by matrices.
- Affine maps are defined by a linear map plus a constant translation.
- Keyed function or permutation families of linear or affine maps cannot be pseudorandom.

Exercises

1. Find all subgroups of $(\mathbb{Z}_{10}, +)$, $(\mathbb{Z}_{11}, +)$ and $(\mathbb{Z}_{11}^*, \cdot)$.
2. Let G be a group of order 54. List the possible orders of elements in G.
3. Consider the map $f : \mathbb{Z}_{19} \to \mathbb{Z}_{19}$ defined by $f(x) = 5x \mod 19$. Show that f is a group homomorphism and even an isomorphism.
4. Let p and q be different prime numbers and $n = pq$. Let e and d be integers such that $ed \equiv 1 \mod (p-1)(q-1)$. Show that $m^{ed} \equiv m \mod n$ for any $m \in \mathbb{Z}_n$.
   Tip: Use Euler's Theorem to show the statement for $m \in \mathbb{Z}_n^*$.
5. Consider the multiplicative group $G = \mathbb{Z}_{23}^*$. Compute the order of $2 \mod 23$. What is the maximum order of elements in G? Find a generator of G.
6. Check whether $2 \mod 19$ and $5 \mod 19$ are generators of the multiplicative group $\mathbb{Z}_{19}^*$ and determine their order.
7. Show that $\mathbb{Z}_n \times \mathbb{Z}_n$ is not cyclic for $n \geq 2$.
   Tip: Verify that ord(g) $\mid n$ for all $g \in \mathbb{Z}_n \times \mathbb{Z}_n$.
8. Find all abelian groups of order 8, up to isomorphism. Which of them are cyclic?

9. Determine the decompositions of $\mathbb{Z}_{12}^*$ and $\mathbb{Z}_{23}^*$ as a product of additive cyclic groups.
10. Which residue classes are generators of the *additive* group $\mathbb{Z}_n$?
11. Let $n = 247 = pq$. Find the factors p and q and solve the simultaneous congruences $k = 7 \mod p$ and $k = 2 \mod q$ using the Chinese Remainder Theorem.
12. Let R_1 and R_2 be rings. Why is the product ring $R_1 \times R_2$ never a field, even if R_1 and R_2 are fields?
    Tip: Consider the idempotent elements $(1, 0)$ and $(0, 1)$.
13. Let $f, g \in K[x]$. Show the product rule
    $$D(f \cdot g) = D(f) \cdot g + f \cdot D(g).$$
    Tip: Use the linearity of the derivative D to reduce to the case $f = x^n$ and $g = x^m$.

14. Determine the number of elements of the following residue class rings. Which of the rings are fields?
    (a) $GF(2)[x]/(x^4 + x^2 + 1)$,
    (b) $GF(3)[x]/(x^2 + 1)$,
    (c) $GF(2)[x]/(x^n - 1)$, where $n \in \mathbb{N}$.
15. Let $GF(8) = GF(2)[x]/(x^3 + x + 1)$. Find representatives of x^3, x^4, x^5, x^6, x^7 in $GF(8)$ of degree less than 3.
16. Find an irreducible polynomial over $GF(2)$ of degree 6.
17. $GF(2^8)$ is the splitting field of $f(x) = x^{256} - x$. Use SageMath to factor $f(x)$ over $GF(2)$ and identity the irreducible factor $g(x) = x^8 + x^4 + x^3 + x + 1$ used to define the AES field.
18. Find explicit descriptions of all subfields of $GF(256)$.
19. Define $GF(2^8)$ using $g(x)$ as above. Which polynomial $f(x)$ corresponds to the byte 02 (hexadecimal notation)? Determine a polynomial $h(x)$ which is inverse to $f(x) \mod g(x)$ and give its hexadecimal representation.
20. Consider the bit permutation $f : GF(2)^8 \to GF(2)^8$ described by (3 1 8 2 5 4 6 7). Determine the inverse bit-permutation f^{-1} and the matrices which represent f and f^{-1}.
21. Show that the following matrix A is unitary and find the inverse matrix A^{-1}:
    $$A = \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}.$$
22. Let f be an affine map given by $f(x) = Ax + b$. Give a necessary and sufficient condition for f being bijective and a formula for f^{-1}.
23. Why is the following map $f : GF(2)^3 \to GF(2)^3$ affine and invertible:
    $$f(x_1, x_2, x_3) = (x_1 + x_2 + x_3 + 1,\ x_1 + x_2,\ x_2 + x_3 + 1)?$$
    Determine the matrix and the translation vector. Compute the inverse map f^{-1}.

24. Let $f : GF(2)^3 \to GF(2)^3$ be a linear map with the following input vectors v_i and output vectors $w_i = f(v_i)$. Determine the matrix which corresponds to f.
$$v_1 = \begin{pmatrix}0\\1\\0\end{pmatrix},\ w_1 = \begin{pmatrix}1\\0\\0\end{pmatrix},\quad v_2 = \begin{pmatrix}0\\0\\1\end{pmatrix},\ w_2 = \begin{pmatrix}1\\0\\1\end{pmatrix},\quad v_3 = \begin{pmatrix}1\\1\\1\end{pmatrix},\ w_3 = \begin{pmatrix}1\\1\\1\end{pmatrix}.$$

25. Let $V = GF(2^8)$. How can you describe a) $GF(2^8)$-linear maps and b) $GF(2)$-linear maps on V? How many different maps exist in case a) and in case b)?

26. Suppose all maps F_k of a keyed function family F are linear. How can an adversary easily win the prf distinguishability experiment? This shows that F is not a pseudorandom function.
Tip: Choose an all-zero input.

Chapter 5

Block Ciphers

A block cipher is a family of permutations which is designed to behave like a pseudorandom permutation. Block ciphers operate on binary strings of fixed length, but in combination with an operation mode they define a variable-length encryption scheme.

In Section 5.1 of this chapter, we study two important construction methods of block ciphers: substitution-permutation networks and Feistel ciphers. Section 5.2 deals with the Rijndael block cipher, which has been adopted as the Advanced Encryption Standard (AES).

Block ciphers and the AES algorithm are dealt with in all modern cryptography textbooks, for example [**PP10**]. A detailed description of the design of AES can be found in [**DR02**].

5.1. Constructions of Block Ciphers

A block cipher is a keyed family of permutations (see Definition 2.46):

$$E : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l.$$

Currently, a block length of $l = 128$ bits and key lengths between $n = 128$ and $n = 256$ bits are widely used. The key space is large enough to prevent brute-force attacks against uniform random keys. For each key k, a permutation E_k of $\{0,1\}^l$ has to be defined. Because of computing and storage limitations, it is impossible to use a table of values and E_k should rather be defined by an efficient algorithm. In practice, one combines the following two types of operations: *linear or affine mixing maps* on full blocks of length l and *nonlinear S-Boxes* on short segments of a block.

- Mixing maps permute all bits of a block. Typically, *bit permutations* are used which are very efficient and spread input changes throughout a block. General linear or affine mixing maps are also used. The required properties (in particular bijectivity) can easily be checked and the computations are very efficient. Furthermore, linear and affine maps can achieve *diffusion* if the map is appropriately chosen: small input changes, say only one bit, affect a whole block and result in large output changes. Note that diffusion is a necessary property of pseudorandom permutations: the output should completely change, even if only a few input bits are modified. Otherwise, an adversary could distinguish E_k from a random permutation.

- S-Boxes are nonlinear, random looking maps which are applied in parallel to short segments of a block. For a small number of input values, for example 8 bits with $2^8 = 256$ values, the S-Box transformation can be defined explicitly by a table. The S-Box needs to be carefully defined and must be highly *nonlinear*.

The combination of linear mixing maps and nonlinear S-Boxes, applied in several rounds, can achieve *confusion*, which makes the relationship between the ciphertext and the key complex and involved. Confusion makes it very hard to find the key or the decryption function, even if many plaintext/ciphertext pairs are known to an adversary.

The properties of confusion and diffusion and their role in the construction of secure ciphers were first described by *Claude Shannon* in 1949 [**Sha49**].

Confusion and diffusion can be achieved by a *substitution-permutation network*. Such a network consists of a number of rounds in which a plaintext block is transformed into a ciphertext block. Each round consists of the following operations (see Figure 5.1):

(1) Add a round key to the data block. The round key is derived from the encryption key and ensures that the transformation depends on the key.

(2) Split the block into smaller segments and apply a nonlinear S-Box (*substitution*) to each of the segments.

(3) Apply a bit *permutation* or, more generally, a linear or affine mixing map to the full data block.

Each operation in a substitution-permutation network has to be invertible so that a unique decryption map exists. Typical networks have at least ten rounds.

Another widely used construction is *Feistel networks*. They are also based on nonlinear S-boxes and linear mixing operations, but split a block into a left and right half. Feistel networks use a round function that depends on a round key and operates on *one half* of the data block (see Figure 5.2).

Let (L_0, R_0) be plaintext block of length $2t$. Define

$$(L_i,\, R_i) = (R_{i-1},\, L_{i-1} \oplus f_{k_i}(R_{i-1})) \qquad \text{for } i = 1, 2, \dots, r.$$

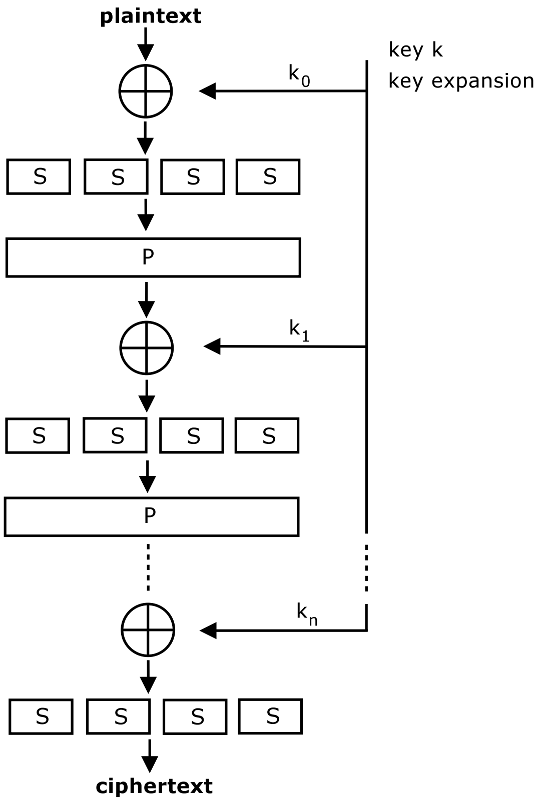


Figure 5.1. Substitution-permutation network: add a round key k_i, apply the S-Box S and the permutation P in n rounds.

After r rounds and a final permutation one obtains the ciphertext of the Feistel cipher:

$$E_k(L_0, R_0) = (R_r, L_r).$$

Now let (R_r, L_r) be a ciphertext block. Then define

$$(R_{i-1},\, L_{i-1}) = (L_i,\, R_i \oplus f_{k_i}(L_i)) \quad \text{for } i = r,\, r-1,\, \dots,\, 1.$$

Note that encryption and decryption use the same transformation. Applying r rounds of the Feistel network and a final permutation recovers the plaintext:

$$D_k(R_r, L_r) = (L_0, R_0).$$

The round function f depends on a round key k_i, and f_{k_i} is usually defined by S-boxes and bit permutations (or affine operations), similar to a substitution-permutation network. However, the round function f only operates on one half of a block and, due to the Feistel network construction, f does not have to be bijective. One can show that a Feistel cipher with at least 3 rounds is a *pseudorandom permutation* if the underlying round function is a *pseudorandom function*. Typical Feistel ciphers have 16 rounds.

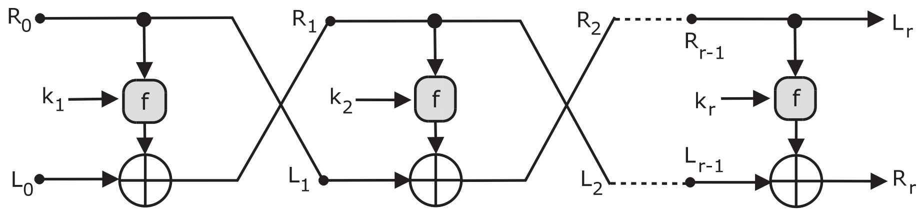


Figure 5.2. Feistel network with *r* rounds.

Many block ciphers are based on Feistel networks, for example the former encryption standard DES, but also modern block ciphers such as Twofish.

5.2. Advanced Encryption Standard

After a 4-year standardization process, the American National Institute of Standards and Technology (NIST) adopted the block cipher Rijndael (designed by J. Daemen and V. Rijmen) as the *Advanced Encryption Standard* (AES) in 2001 [**FIP01**]. The cipher is now widely used by security applications and network protocols. Rijndael is fast on both software and hardware (also on 8-bit processors), easy to implement and requires little memory. Throughputs of several hundred MBit/s in software and several GBit/s in hardware are easily possible. There are a number of alternatives, for example Twofish, Serpent, Camellia and others, but at the time of this writing AES is the dominating block cipher.

The standardized AES cipher has a block length of 128 bits and a 128-, 192- or 256-bit key length. The original Rijndael cipher supports additional block lengths and key lengths. The cipher is a *substitution-permutation network*, and the number of rounds depends on the key size; the cipher has either 10 (for 128-bit keys), 12 (for 192-bit keys), or 14 rounds (for 256-bit keys). These settings allow for a very efficient operation. Furthermore, the block and key lengths prevent simple brute-force attacks and lookup tables.

We denote the AES encryption function by $f_k : \{0,1\}^{128} \to \{0,1\}^{128}$. The encryption function operates on the 128-bit *state*. The state is arranged in a 4×4 matrix over $GF(2^8)$ by writing the bytes $p_0, p_1, \dots, p_{16}$ into the columns of a matrix:

$$\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix}.$$

Each byte is interpreted as an element of the field

$$GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$$

(compare Example 4.77). The byte $(b_7 \dots b_1 b_0)$ corresponds to the residue class $b_7 x^7 + \dots + b_1 x + b_0 \mod (x^8 + x^4 + x^3 + x + 1)$.

Example 5.1. Let $m = 10\dots0$ be a 128-bit input block. The byte 80 $= 1000\,0000$ corresponds to $x^7 \mod x^8 + x^4 + x^3 + x + 1$. Obviously, the zero byte corresponds to the zero polynomial. Thus, m is represented by the following 4×4 matrix over $GF(2^8)$:

$$\begin{pmatrix} x^7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \qquad \Diamond$$

The AES encryption function f_k takes the plaintext as input state and transforms the state in successive rounds. Each round consists of several steps: the nonlinear substitution step (SubBytes), two linear mixing steps (ShiftRows, MixColumns) and the affine AddRoundKey step. The final state is output. Each step is invertible and the decryption function f_k^{-1} is given by composing the inverse steps in reverse order.

The following pseudocode gives a high-level description of f_k. The SubBytes, ShiftRows, MixColumns operations and the KeyExpansion step are described below.

```
Rijndael(State, CipherKey)
{
   KeyExpansion(CipherKey, ExpandedKey)
   AddRoundKey(State,ExpandedKey[0])
   for(i = 1; i < Nr ; i++)  {     // Nr is either 10, 12 or 14
      // Round i
      SubBytes(State)
      ShiftRows(State)
      MixColumns(State)
      AddRoundKey(State,ExpandedKey[i])
   }
   // Final Round
   SubBytes(State)
   ShiftRows(State)
   AddRoundKey(State,ExpandedKey[Nr])
}
```

First, we consider the S-Box SubBytes which is the only non-affine component of AES. The S-Box function $S_{RD} : GF(2^8) \to GF(2)^8$ is applied to each byte of the state individually (see Figure 5.3).

$$\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \xrightarrow{\text{SubBytes}} \begin{pmatrix} S_{RD}(p_0) & S_{RD}(p_4) & S_{RD}(p_8) & S_{RD}(p_{12}) \\ S_{RD}(p_1) & S_{RD}(p_5) & S_{RD}(p_9) & S_{RD}(p_{13}) \\ S_{RD}(p_2) & S_{RD}(p_6) & S_{RD}(p_{10}) & S_{RD}(p_{14}) \\ S_{RD}(p_3) & S_{RD}(p_7) & S_{RD}(p_{11}) & S_{RD}(p_{15}) \end{pmatrix}$$

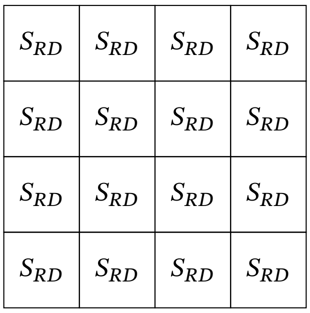


Figure 5.3. The nonlinear S-Box operates on each byte of the state array individually.

S_{RD} is defined by multiplicative inversion in the field $GF(2^8)$ followed by an affine transformation of the vector space $GF(2)^8$. The definition

$$GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$$

gives a $GF(2)$-linear isomorphism between $GF(2^8)$ and $GF(2)^8$, so that elements in $GF(2^8)$ correspond to a binary word of length 8 (one byte).

$$S_{RD}(a) = \begin{cases} Aa^{-1} + b & \text{for } a \neq 0, \\ b & \text{for } a = 0, \end{cases}$$

$$A = \begin{pmatrix} 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \end{pmatrix}, \qquad b = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}.$$

Since $0 \in GF(2^8)$ is not invertible, one extends the inversion by mapping 0 to 0. The extended inversion map is a bijection on $GF(2^8)$ and can also be described by the monomial map $i(a) = a^{254}$. In fact, Euler's Theorem can be applied to the multiplicative group $GF(2^8)^*$ of units. This yields $a^{255} = 1$ and hence $a^{-1} = a^{254}$ for all $a \neq 0$. The composition of the inversion map i and the affine transformation $f(a) = Aa + b$ can be represented by a polynomial over $GF(2^8)$ (see [**DR02**]):

$$S_{RD}(a) = f(i(a)) = \mathtt{05} \cdot a^{254} + \mathtt{09} \cdot a^{253} + \mathtt{F9} \cdot a^{251} + \mathtt{25} \cdot a^{247} + \mathtt{F4} \cdot a^{239} \\ + \mathtt{01} \cdot a^{223} + \mathtt{B5} \cdot a^{191} + \mathtt{8F} \cdot a^{127} + \mathtt{63}.$$

This shows that S_{RD} has a complex algebraic expression over $GF(2^8)$. One can also consider S_{RD} as an $(8, 8)$-vectorial Boolean function and compute the algebraic normal form (see Section 1.1) of each of its components. The algebraic degree of the Boolean functions is 7, which demonstrates a high algebraic complexity over $GF(2)$.

Note that implementations of AES do not use the algebraic definition of S_{RD}, but a lookup table instead. This requires only 256 bytes of memory.

Example 5.2. We use SageMath, construct an AES object called `sr` and print out the hexadecimal S-Box values:

```
sage: sr = mq.SR(10, 4, 4, 8, star=True, allow_zero_inversions=True, aes_mode=True)
sage: S=sr.sbox()
sage: for i in range(0,256):
          print "{:02X}".format(S[i]),
63 7C 77 7B F2 6B 6F C5 30 01 67 2B FE D7 AB 76 CA 82 C9 7D FA 59 47 F0
AD D4 A2 AF 9C A4 72 C0 B7 FD 93 26 36 3F F7 CC 34 A5 E5 F1 71 D8 31 15
04 C7 23 C3 18 96 05 9A 07 12 80 E2 EB 27 B2 75 09 83 2C 1A 1B 6E 5A A0
52 3B D6 B3 29 E3 2F 84 53 D1 00 ED 20 FC B1 5B 6A CB BE 39 4A 4C 58 CF
D0 EF AA FB 43 4D 33 85 45 F9 02 7F 50 3C 9F A8 51 A3 40 8F 92 9D 38 F5
BC B6 DA 21 10 FF F3 D2 CD 0C 13 EC 5F 97 44 17 C4 A7 7E 3D 64 5D 19 73
60 81 4F DC 22 2A 90 88 46 EE B8 14 DE 5E 0B DB E0 32 3A 0A 49 06 24 5C
C2 D3 AC 62 91 95 E4 79 E7 C8 37 6D 8D D5 4E A9 6C 56 F4 EA 65 7A AE 08
BA 78 25 2E 1C A6 B4 C6 E8 DD 74 1F 4B BD 8B 8A 70 3E B5 66 48 03 F6 0E
61 35 57 B9 86 C1 1D 9E E1 F8 98 11 69 D9 8E 94 9B 1E 87 E9 CE 55 28 DF
8C A1 89 0D BF E6 42 68 41 99 2D 0F B0 54 BB 16
```

For example, $S_{RD}(\mathtt{00}) = b = \mathtt{63}$ and $S_{RD}(\mathtt{01}) = \mathtt{7C}$. $\Diamond$

One of the main design criteria of the S-Box is its *nonlinearity*. We have seen above that S_{RD} is nonlinear and its algebraic degree is high. In addition, it is impossible to *approximate* S_{RD} by affine functions. One can show that any linear combination (XOR) of input and output bits of the S-Box gives the correct value for at least 112 and at most 144 of 256 input values. Note that the expected number of matches for a random XOR combination is 128. The correlation between the S-Box and all affine functions is therefore low, which protects the cipher against a linear cryptanalysis.

Another design aspect is the *differential properties* of the S-Box. It can be shown that for any fixed pair of input-output differences, at most 4 out of 256 values propagate the given differences. This prevents a differential cryptanalysis of the cipher.

Next, we look at the diffusion layer, which is implemented by the linear mixing maps *ShiftRows* and *MixColumns*. Both are $GF(2^8)$-linear operations on the state matrix.

ShiftRows is a bit permutation and rotates the bytes in the second, third and fourth row to the left. The first row is left unchanged, the bytes in the second row are rotated by one position, bytes in the third row are rotated by two positions and bytes in the fourth row are rotated by three positions (see Figure 5.4). Clearly, ShiftRows can be inverted by a corresponding circular right shift.

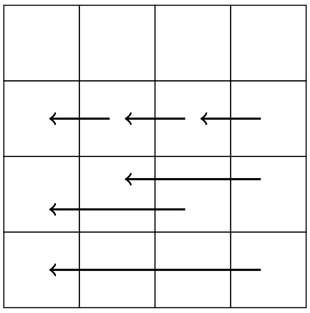


Figure 5.4. ShiftRows rotates the bytes in the rows by zero, one, two and three positions, respectively.

$$\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \xrightarrow{\text{ShiftRows}} \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_5 & p_9 & p_{13} & p_1 \\ p_{10} & p_{14} & p_2 & p_6 \\ p_{15} & p_3 & p_7 & p_{11} \end{pmatrix}.$$

MixColumns transforms the columns of the state matrix by a $GF(2^8)$-linear map. One multiplies a constant 4×4 matrix M over $GF(2^8)$ by the column vectors of the state (see Figure 5.5). The matrix is regular so that the operation can be inverted (see Section 0.4 where the inverse matrix is computed). The MixColumns matrix was carefully chosen to have good diffusion properties. If $v \in GF(2^8)^4$ is a nonzero column vector, then the number of nonzero bytes of v plus the number of nonzero bytes of Mv is at least 5. This can be shown using linear codes (see Example 15.19 (2)).

$$\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \xrightarrow{\text{MixColumns}} \underbrace{\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}}_{M} \cdot \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix}$$

In the *AddRoundKey* step, every bit of the state matrix is XORed with the round key k_i. The round keys have the same length as the state (128 bits) and are computed in the *KeyExpansion* step, as explained below.

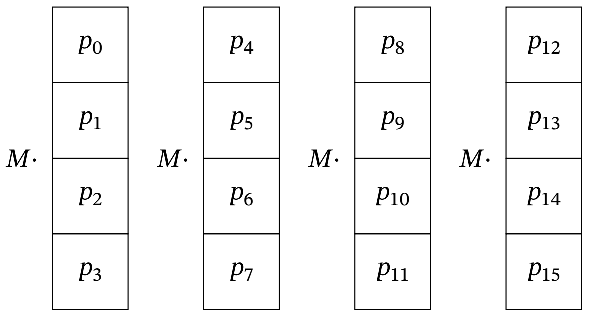


Figure 5.5. The MixColumns operation: each column is transformed by a fixed matrix M.

$$\begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \xrightarrow{\text{AddRoundKey}} \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} \bigoplus k_i.$$

The operations were designed such that two Rijndael rounds (SubBytes, ShiftRows, MixColumns, AddRoundKey) already provide sufficient diffusion. After two rounds, every output bit depends on all input bits, and a change in one input bit changes about half of all output bits.

Finally, we explain AES key scheduling. The main design criteria for the *key expansion* step were *efficiency, symmetry elimination, diffusion of the key and nonlinearity*. The nonlinearity is intended to protect the cipher against *related-key attacks* (compare Remark 2.44).

We begin with 128-bit keys (see Figure 5.6). In this case, the AES algorithm has ten rounds, and eleven 128-bit round keys $k_0, k_1, \dots, k_{10}$ are required. The subkeys are stored in 44 words $W_0, W_1, \dots, W_{43} \in GF(2^8)^4$ of length 32 bits. Let

$$sh : GF(2^8)^4 \to GF(2^8)^4$$

be the rotation by one byte position to the left, i.e., $sh(p_0, p_1, p_2, p_3) = (p_1, p_2, p_3, p_0)$. We write $\mathbf{S}$ for the function which applies the S-Box S_{RD} to all four components of a vector in $GF(2^8)^4$. This function ensures that the key schedule is *nonlinear*. The symmetry of $\mathbf{S}$ is eliminated by *round constants*:

$$RC_j = x^{j-1} \bmod x^8 + x^4 + x^3 + x + 1 \in GF(2^8) \text{ for } j \geq 1.$$

The 128-bit AES key k defines the initial round key $k = k_0 = W_0||W_1||W_2||W_3$. The next round key $k_1 = W_4||W_5||W_6||W_7$ is computed as follows:

$$\begin{aligned} W_4 &= W_0 \oplus \mathbf{S}(sh(W_3)) \oplus (RC_1, 0, 0, 0), & W_5 &= W_1 \oplus W_4, \\ W_6 &= W_2 \oplus W_5, & W_7 &= W_3 \oplus W_6. \end{aligned}$$

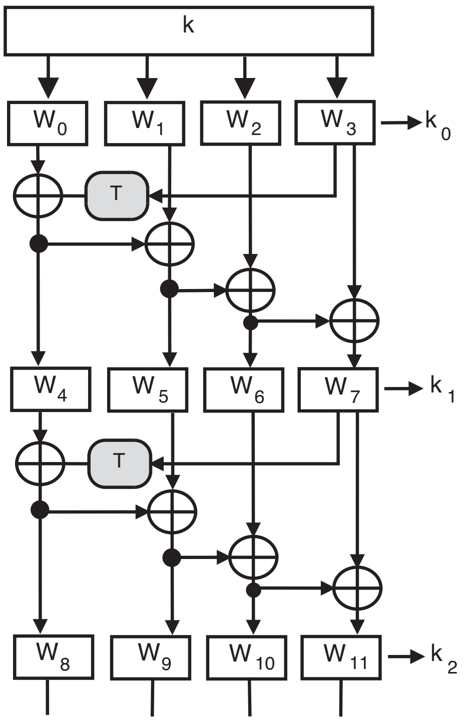


Figure 5.6. The first two rounds of 128-bit AES key scheduling. T maps the word W_{4i-1} to $\mathbf{S}(sh(W_{4i-1})) \oplus (RC_i, 0, 0, 0)$. This is basically a byte-wise SubBytes operation, but involves an additional rotation and a translation by a round constant.

The following round keys are constructed analogously (increment the index of RC by 1 and increase all other indices by 4). For $i = 2, \dots, 10$ one defines:

$$W_{4i} = W_{4i-4} \oplus \mathbf{S}(sh(W_{4i-1})) \oplus (RC_i, 0, 0, 0), \quad W_{4i+1} = W_{4i-3} \oplus W_{4i},$$
$$W_{4i+2} = W_{4i-2} \oplus W_{4i+1}, \quad W_{4i+3} = W_{4i-1} \oplus W_{4i+2}.$$

The round keys are given by

$$k_i = W_{4i} \| W_{4i+1} \| W_{4i+2} \| W_{4i+3} \text{ for } i = 0, 1, \dots, 10.$$

Note that the construction of the first word of each round key uses the nonlinear S-Box, while the other three words are defined by XORing two preceding words.

We skip 192-bit keys and look at 256-bit keys. In this case, fifteen 128-bit subkeys $k_0, k_1, \dots, k_{14}$ are needed. The key schedule generates 60 words $W_0, W_1, \dots, W_{59}$ in $GF(2^8)^4$ and the round keys are given by

$$k_i = W_{4i} \| W_{4i+1} \| W_{4i+2} \| W_{4i+3} \text{ for } i = 0, 1, \dots, 14.$$

A 256-bit AES key k defines the first eight words $W_0, W_1, \dots, W_7$. The next eight words $W_8, W_9, \dots, W_{15}$ are computed as follows:

$$\begin{aligned} W_8 &= W_0 \oplus \mathbf{S}(sh(W_7)) \oplus (RC_1, 0, 0, 0), & W_9 &= W_1 \oplus W_8, \\ W_{10} &= W_2 \oplus W_9, & W_{11} &= W_3 \oplus W_{10}, \\ W_{12} &= W_4 \oplus \mathbf{S}(W_{11}), & W_{13} &= W_5 \oplus W_{12}, \\ W_{14} &= W_6 \oplus W_{13}, & W_{15} &= W_7 \oplus W_{14}. \end{aligned}$$

The following eight words are defined analogously (increment the index of RC by 1 and increase all other indices by 8), until all 60 words have been defined. Again, the first word of each round key is defined by a nonlinear operation, which in turn affects all subsequent words.

5.3. Summary

- A block cipher is a family of permutations. It should have good diffusion and confusion properties.
- The main building blocks of block ciphers are linear or affine mixing maps and nonlinear S-Boxes.
- Substitution-permutation networks and Feistel networks are the most important methods for constructing block ciphers.
- The Rijndael block cipher was adopted as the Advanced Encryption Standard (AES) and is widely used in practice.
- AES is a substitution-permutation network with a block length of 128 bits. The cipher operates on a 4×4 state matrix over $GF(2^8)$ and is defined by a sequence of KeyExpansion, AddRoundKey, SubBytes, ShiftRows and MixColumns steps.
- The AES SubBytes S-Box is the only nonlinear operation. It is defined by a multiplicative inversion in $GF(2^8)$, followed by an affine map over $GF(2)$.
- The AES key schedule takes a 128-, 192- or 256-bit key as input and generates 128-bit round keys by linear and nonlinear operations.
- AES is resistant to various attacks and modeled as a family of pseudorandom permutations.

Exercises

1. Consider Feistel ciphers. Use the formulas in Section 5.1 to show that D_k recovers the plaintext.
2. Verify the following inverses in $GF(2^8)$ (in hexadecimal notation):

$$\texttt{01}^{-1} = \texttt{01},\ \texttt{02}^{-1} = \texttt{8D},\ \texttt{03}^{-1} = \texttt{F6}.$$

Then compute $S_{RD}(\texttt{00})$, $S_{RD}(\texttt{01})$, $S_{RD}(\texttt{02})$ and $S_{RD}(\texttt{03})$.

3. Let $n \in \mathbb{N}$. Show that $f(x) = x^{(2^n)}$ is a $GF(2)$-linear map on $GF(2^8)$, whereas $f(x) = x^{254}$ is not linear.
4. Describe the inverse S-Box S_{RD}^{-1}.
5. How can the multiplication of 8-bit strings by 01, 02 and 03 be efficiently implemented? What is an advantage of the MixColumns matrix? What can be said about its inverse matrix?
6. Show that the MixColumn matrix and all submatrices are nonsingular over $GF(2^8)$. One can show that this ensures good diffusion properties (see Example 15.19 (2)).
7. Give a high-level (pseudocode) description of the AES decryption function f_k^{-1}.
8. Suppose a 128-bit AES key k and a plaintext m are given:

   $k =$ 01 00 00 00 00 00 00 00 00, $m =$ 80 00 00 00 00 00 00 00 00.

   (a) Find the round keys k_0 and k_1.
   (b) Use SageMath to compute all round keys and encrypt the input block m.
   Tip: The following SageMath function may be used:

```
sage: sr = mq.SR(10, 4, 4, 8, star=True,
          allow_zero_inversions=True, aes_mode=True)
sage: def aesenc(p,k):
           # Add k=key0
           print sr.hex_str_vector(k)
           p=p+k;
           # Rounds 1-9
           for i in range(1,10):
                 p=sr.sub_bytes(p)
                 p=sr.shift_rows(p)
                 p=sr.mix_columns(p)
                 k=sr.key_schedule(k, i)
                 p=p+k
                 print sr.hex_str_vector(k)
           # Round 10
           p=sr.sub_bytes(p)
           p=sr.shift_rows(p)
           k=sr.key_schedule(k, 10)
           print sr.hex_str_vector(k)
           p=p+k
           print "Output" + sr.hex_str_vector(p)
           return p
```

Define $K = GF(2^8)$ and initialize 4×4 matrices M and Key. Only the upper left entries of these matrices are nonzero in this exercise.

```
sage: K.<a>=GF(2^8, name='a', modulus=x^8+x^4+x^3+x+1)
sage: M=sr.state_array(); M[0,0]=a^7
sage: Key=sr.state_array(); Key[0,0]=1
```

9. Assume that a modified AES block cipher lacks all *ShiftRows* and *MixColumns* operations. Can this cipher be a pseudorandom permutation? What if only one of these operations is missing?
10. Suppose that the multiplicative inversion is omitted in the S-Box of a modified AES block cipher. Can this cipher be a pseudorandom permutation?
11. What is more important for a cipher: the nonlinearity of encryption or the nonlinearity of the key schedule?
12. Suppose a 256-bit AES key k is all-zero. Find the round keys k_0, k_1, k_2 and k_3.

Chapter 6

Stream Ciphers

Symmetric ciphers can be divided into block ciphers and stream ciphers. Some operation modes turn block ciphers into stream ciphers, for example the counter mode, but this chapter focuses on dedicated stream ciphers that are constructed as keystream generators. Stream ciphers are usually very fast, even on restricted hardware. They were used a lot in the past, for example to protect network communication, but in many cases have been replaced by the AES block cipher.

Section 6.1 deals with synchronous stream ciphers and self-synchronizing stream ciphers and presents the block cipher modes OFB and CFB. We introduce two classical stream ciphers in Sections 6.2 and 6.3, linear feedback shift registers (LFSRs) and the RC4 cipher, and outline their vulnerabilities. In Section 6.4, we provide an example of a new stream cipher family, Salsa20, and the related ChaCha family.

Stream ciphers are contained in most cryptography textbooks, for example [**PP10**] and [**KL15**]. We also recommend the handbook [**MvOV97**]. Further details on the design of new stream ciphers can be found in [**RB08**] and at the eSTREAM project (`http://www.ecrypt.eu.org/stream/`).

6.1. Definition of Stream Ciphers

In Section 2.8 we introduced pseudorandom generators of fixed output length. A *stream cipher* is essentially a pseudorandom generator that outputs keystream bits, but our definition differs in two ways: the output length is not fixed and the keystream is computed recursively using an *internal state* and the key. The initial state is derived from a key and an initialization vector.

We use the term *stream cipher* for an encryption scheme that is based on a *keystream generator*. Encryption is defined by XORing the plaintext with the keystream, i.e., by bitwise addition of the plaintext and the keystream. Other encryption functions which combine several plaintext and keystream bits to produce ciphertext are also possible. Note the difference to block ciphers (Chapter 5), which process larger blocks of plaintext (e.g., 128 bits). Stream ciphers process small plaintext blocks, e.g., only one bit, and the keystream varies as the plaintext is processed. Two types of cipher streams can be distinguished. In *synchronous stream ciphers*, the keystream depends only on the key and the internal state of the generator. *Self-synchronizing stream ciphers*, on the other hand, use the previous ciphertext bits to generate the keystream. Below we assume that encryption and decryption is given by binary addition.

Definition 6.1. A *synchronous stream cipher* is an encryption scheme defined by the following spaces and polynomial-time algorithms:

- The plaintext space and the ciphertext space is $\mathcal{M} = \mathcal{C} = \{0,1\}^*$.
- The key generation algorithm $Gen(1^n)$ takes 1^n as input and outputs a key $k \in \{0,1\}^n$ as well as an initialization vector IV.
- The initialization algorithm $Init(k, IV)$ takes k and IV as input and outputs an initial state st_1.
- The keystream generator $G = G(k, st)$ takes k and st as input and recursively computes l-bit output words y_1, y_2, ... called a *keystream*. The next state function $f(k, st)$ takes k and st as input and updates the state st.

  $$y_i = G(k,\ st_i) \text{ and } st_{i+1} = f(k,\ st_i) \text{ for } i \geq 1.$$

- Encryption of a plaintext $(m_1, m_2, \dots)$ and decryption of a ciphertext $(c_1, c_2, \dots)$ are defined by XORing each input word of length l with the corresponding keystream word (see Figure 6.1).

  $$c_i = m_i \oplus y_i \text{ and } m_i = c_i \oplus y_i \text{ for } i \geq 1.$$

  If the last plaintext or ciphertext word is shorter than l bits, then only the first (most significant) bits of the keystream word are used. ◊

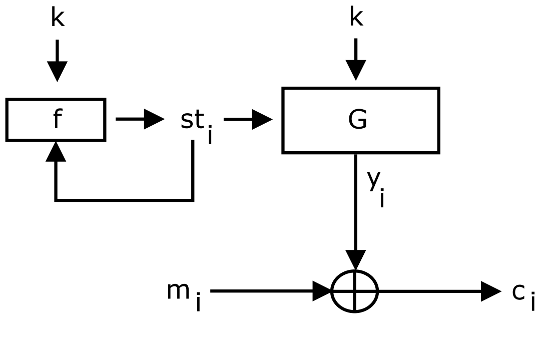


Figure 6.1. Keystream generation and encryption using a synchronous stream cipher.

The keystream of a synchronous stream cipher does not depend on the plaintext or the ciphertext. The sender and receiver must be *synchronized* and use the same state for the decryption to be successful.

Example 6.2. The *Output Feedback* (*OFB*) *mode* (see [**Dwo01**]) turns a block cipher into a synchronous stream cipher. Let $F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ be a keyed family of functions, for example a block cipher. A uniform random key $k \overset{\$}{\leftarrow} \{0,1\}^n$ and a uniform initialization vector $IV \overset{\$}{\leftarrow} \{0,1\}^l$ are chosen. The initial state is $st_1 = y_0 = IV$, and we recursively generate keystream words of length l by applying F_k to the state (see Figure 6.2). The keystream is also used to update the state.

$$y_i = F_k(st_i) = F_k(y_{i-1}) \text{ and } st_{i+1} = y_i \text{ for } i \geq 1.$$

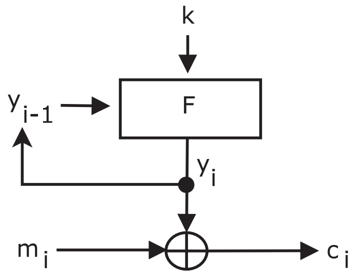


Figure 6.2. OFB mode encryption. The cipher recursively generates keystream words.

Another example of a block cipher turned into a synchronous stream cipher is given by the CTR (counter) mode (see Definition 2.48). In that case, the counter value represents the state. The counter is incremented after each output block. ♢

Self-synchronizing stream ciphers (also called asynchronous stream ciphers or ciphertext autokey) are less common. For example, only two out of 34 submissions to the eSTREAM project were of this type.

Definition 6.3. A *self-synchronizing stream cipher* is an encryption scheme that differs from a synchronous stream cipher (see Definition 6.1) in the following way:

The *state* st_i is represented by $t \geq 1$ preceding *ciphertext words* $(c_{i-t}, \dots, c_{i-1})$, where the initial state is the initialization vector $IV = st_1 = (c_{1-t}, \dots, c_0)$.

$$y_i = G(k, st_i) = G(k, (c_{i-t}, \dots, c_{i-1})),\ c_i = m_i \oplus y_i,\ st_{i+1} = (c_{i+1-t}, \dots, c_i)$$

for $i \geq 1$ (see Figure 6.3 for $t = 1$). ♢

Note that a synchronization is possible after receiving t correct ciphertext words. This does not work for a synchronous stream cipher since its state is recursively updated and cannot be derived from the ciphertext.

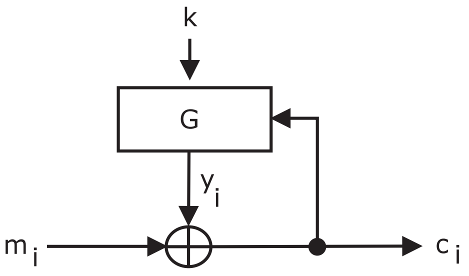


Figure 6.3. Self-synchronizing stream cipher.

Example 6.4. A block cipher in *Cipher Feedback* (*CFB*) *mode* (see [**Dwo01**]) gives rise to a self-synchronizing stream cipher. Let $F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ be a keyed family of functions, for example a block cipher. Choose a uniform key $k \overset{\$}{\leftarrow} \{0,1\}^n$ and a uniform initialization vector $IV \overset{\$}{\leftarrow} \{0,1\}^l$. The initial state is $st_1 = c_0 = IV$. Let m_i be the i-th plaintext word of length l. Define the keystream and the ciphertext words by

$$y_i = F_k(c_{i-1}) \text{ and } c_i = m_i \oplus y_i \text{ for } i \geq 1.$$

The keystream depends on the preceding ciphertext word ($t = 1$, see Figure 6.4). There are also s-bit CFB modes where words of length $s \leq l$ are processed. ◊

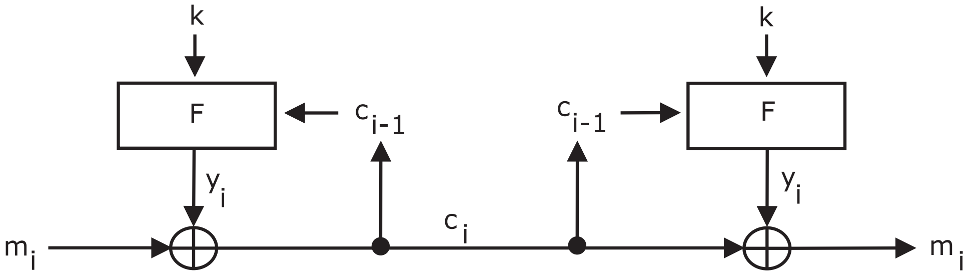


Figure 6.4. CFB mode encryption and decryption.

There is no obvious *security definition* for stream ciphers. In the following, we discuss possible definitions for synchronous stream ciphers.

If the cipher does not depend on an IV, then we can refer to Definition 2.32 of a *pseudorandom generator*. The generated keystream y_1, y_2, y_3, ... should be pseudorandom; i.e., a probabilistic polynomial-time adversary, who does not know the secret key, should not be able to distinguish the keystream of polynomial length from a truly random bit sequence.

For an IV-dependent stream cipher $G(k, IV)$, the minimum requirement would be the pseudorandomness of the keystream (as above) for random IVs, where the IV is known to an adversary.

For a stronger security definition, we let the adversary choose an IV and the output length. They are given either the associated keystream of the chosen length or a random bit sequence of the same length. The adversary's task is to distinguish between the two cases.

The cipher is secure if $G(k, IV)$ is a *pseudorandom function* (see Definitions 2.38 and 2.39). An IV-dependent stream cipher can in fact be viewed as a *family of functions*, which is parametrized by a key and maps an IV to a keystream. Refer to [**BG07**] and [**Zen07**] for a discussion of this topic.

Yet another security issue not addressed here are *related-key attacks* (see Remark 2.44) against stream ciphers.

6.2. Linear Feedback Shift Registers

A classical and very efficient way to generate output bits from an initial state is to use a *linear feedback shift register* (LFSR). An LFSR of degree n consists of n binary registers that form the *state*. The state is updated by shifting all registers to the right (see Figure 6.5). The rightmost bit leaves the state and becomes an output bit. The new leftmost bit is an XOR combination of state bits, and fixed feedback coefficients determine whether or not a particular position of the state is tapped.

Definition 6.5. A linear feedback shift register (LFSR) of degree n (or length n) is defined by feedback coefficients $c_1, c_2, \dots, c_n \in GF(2)$. The initial state is an n-bit word $st = (s_{n-1}, \dots, s_1, s_0)$ and new bits are generated by the recursion

$$s_j = c_1 s_{j-1} + c_2 s_{j-2} + \cdots + c_n s_{j-n} \bmod 2 \quad \text{for } j \geq n.$$

At each iteration step (clock tick), the state st is updated from $(s_{j-1}, \dots, s_{j-n})$ to $(s_j, s_{j-1}, \dots, s_{j-n+1})$, i.e., by shifting the register to the right. The rightmost bit s_{j-n} is output. The output of an LFSR is called a *linear recurring sequence*. ◊

Obviously, an LFSR first outputs the initial state $s_0, s_1, \dots, s_{n-1}$ and subsequently the new feedback bits $s_n, s_{n+1}, \dots$. At each step, the state vector st is updated to $A \cdot st$, where A is an $n \times n$ matrix over $GF(2)$ and st is viewed as a column vector. The initial state is $st = (s_{n-1}, \dots, s_1, s_0)^T$.

$$A = \begin{pmatrix} c_1 & c_2 & \dots & c_n \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ & \dots & & \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ and } A \cdot \begin{pmatrix} s_{j-1} \\ s_{j-2} \\ s_{j-3} \\ \dots \\ s_{j-n} \end{pmatrix} = \begin{pmatrix} s_j \\ s_{j-1} \\ s_{j-2} \\ \dots \\ s_{j-n+1} \end{pmatrix} \text{ for } j \geq n.$$

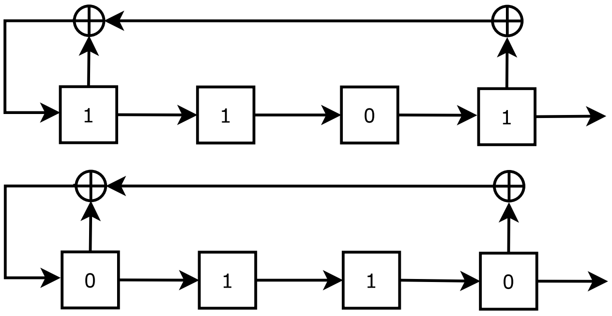


Figure 6.5. Clocking an LFSR of degree 4.

Remark 6.6. The literature has not adopted a unique notation of shift registers and their parameters. LFSRs can also be shifted to the left so that the leftmost bit is output. In this case, the state vector (from left to right) has increasing indices. The initial state is $st = (s_0, s_1, \dots, s_{n-1})$, and the recursion formula as well as the above transition matrix look slightly different. We adopt the notation used by [**MvOV97**]. ◊

It is easy to see that every linear recurring sequence must ultimately be periodic: the feedback coefficients are fixed and the output depends only on the state vector. The state is a binary word of length n, and so there are 2^n possible states.

Definition 6.7. Let $s_0, s_1, \dots$ be a linear recurring sequence. The (*least*) *period* of the sequence is the smallest integer $N \geq 1$ such that

$$s_{j+N} = s_j$$

for all sufficiently large values of j.

Proposition 6.8. *The period of a sequence generated by an LFSR of degree n is at most* $2^n - 1$.

Proof. Consider the sequence of state vectors. If the all-zero state occurs, then the following output is constantly 0 and the period is 1. Otherwise, all state vectors are nonzero. There are $2^n - 1$ nonzero states, and so the period is bounded by this number. □

Example 6.9. Consider an LFSR of degree 4 with feedback coefficients $c_1 = 1, c_2 = 0$, $c_3 = 0$ and $c_4 = 1$. Suppose the initial state is $st = (1, 1, 0, 1)$ (see Figure 6.5). The state is shifted to the right and a new bit is generated by XORing the first and fourth bit of the state. The updated state is $st = (0, 1, 1, 0)$. We continue in this fashion and check that the LFSR assumes all 15 nonzero states. The following output bits are generated: $1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0$ (in this order). The sequence recurs after 15 output bits.

SageMath can also compute the output of LFSRs. The `key` array contains the feedback coefficients and the `fill` array the initial state (in reverse order, so that the leftmost bit is the first output bit of the generator). We generate 20 bits and observe that the output repeats after 15 bits.

```
sage: o = GF(2)(0); l = GF(2)(1)
sage: key = [l,o,o,l]; fill = [l,o,l,l]
sage: s = lfsr_sequence(key,fill,20); s
[1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0]
```

We used the symbols `o` and `l` for $\overline{0} \in GF(2)$ and $\overline{1} \in GF(2)$, respectively. ◊

In the above Example 6.9, the period of the sequence is maximal. In general, the period depends on the initial state and the parameters of an LFSR.

Definition 6.10. Let $c_1, c_2, \dots, c_n$ be the feedback coefficients of an LFSR of degree n. Then $c(x) = 1 + c_1x + c_2x^2 + \dots + c_nx^n \in GF(2)[x]$ is called *connection polynomial* or *feedback polynomial* of the LFSR.

Proposition 6.11. *Let $c(x) = 1 + c_1x + c_2x^2 + \dots + c_nx^n$ be the connection polynomial of an LFSR with associated $n \times n$ matrix*

$$A = \begin{pmatrix} c_1 & c_2 & \dots & c_n \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ & \dots & & \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Then:

(1) *If $c_n = 1$ then $\det(A) = 1$ and A is nonsingular. In this case, the LFSR is called nonsingular.*

(2) *A nonsingular LFSR is invertible, i.e., can run in the reverse direction, and a state uniquely determines the preceding bits.*

(3) *If the LFSR is nonsingular, then an output sequence of period N is purely periodic, i.e., $s_{j+N} = s_j$ for all $j \in \mathbb{N}$.*

(4) *The characteristic polynomial of the matrix A is*

$$p(x) = \det(A - xI_n) = x^n c\left(\frac{1}{x}\right) = x^n + c_1x^{n-1} + \dots + c_{n-1}x + c_n.$$

(5) *The connection polynomial of a nonsingular LFSR satisfies*

$$c(x) = \det(xA - I_n).$$

(6) *The characteristic polynomial $p(x)$ is equal to the minimal polynomial of A and*

$$GF(2)[x]/(p(x)) \cong GF(2)[A],$$

where $GF(2)[A]$ is the commutative subring of matrices over $GF(2)$ that can be written as a sum $a_0I_n+a_1A+a_2A^2+\cdots+a_mA^m$ with $m \in \mathbb{N}$ and $a_0, a_1, \dots, a_m \in GF(2)$.

Proof. We leave it to the reader to prove (1). If A is nonsingular, then A^{-1} exists and

$$A^{-1} \cdot \begin{pmatrix} s_j \\ s_{j-1} \\ s_{j-2} \\ \dots \\ s_{j-n+1} \end{pmatrix} = \begin{pmatrix} s_{j-1} \\ s_{j-2} \\ s_{j-3} \\ \dots \\ s_{j-n} \end{pmatrix}.$$

This shows (2). Since any LFSR is ultimately periodic and nonsingular LFSRs can run in the reverse direction, all output bits of such LFSRs are periodic which proves (3). We now turn to part (4). Since $c(x) = 1 + c_1x + c_2x^2 + \cdots + c_nx^n$ we have

$$x^n c\left(\frac{1}{x}\right) = x^n + c_1x^{n-1} + \cdots + c_{n-1}x + c_n.$$

We prove by induction that this gives the characteristic polynomial of A. If $n = 1$ then $A = (c_1)$, so that $c_1 - x = x + c_1$ is the characteristic polynomial of A over $GF(2)$. Using the hypothesis for LFSRs of degree $n - 1$, we compute the characteristic polynomial $p(x)$ of A:

$$\begin{aligned}
\det(A - xI_n) &= \begin{vmatrix} c_1 + x & c_2 & \dots & c_n \\ 1 & x & 0 & 0 \\ 0 & 1 & x & 0 \\ & \dots & & \\ 0 & 0 & 1 & x \end{vmatrix} \quad \text{(expansion along the last column)} \\
&= c_n \begin{vmatrix} 1 & x & 0 & 0 \\ 0 & 1 & x & 0 \\ & \dots & & \\ 0 & 0 & 0 & 1 \end{vmatrix} + x \begin{vmatrix} c_1 + x & c_2 & \dots & c_{n-1} \\ 1 & x & 0 & 0 \\ & \dots & & \\ 0 & 0 & 1 & x \end{vmatrix} \quad \text{(use hypothesis)} \\
&= c_n + x \cdot (x^{n-1} + c_1x^{n-2} + \cdots + c_{n-2}x + c_{n-1}) \\
&= x^n + c_1x^{n-1} + \cdots + c_{n-1}x + c_n.
\end{aligned}$$

This shows (4). Now we derive (5) from (4):

$$c(x) = c\left(\frac{1}{1/x}\right) = x^n p\left(\frac{1}{x}\right) = x^n \det\left(A - \frac{1}{x}I_n\right) = \det(xA - I_n).$$

It remains to prove (6). In general, the minimal polynomial of a matrix divides the characteristic polynomial (Cayley-Hamilton theorem). Now one can easily see from the definition of A that the first unit vector e_1 is a cyclic vector of A: the vectors e_1, $A \cdot e_1, \dots, A^{n-1}e_1$ span $GF(2)^n$. If the minimal polynomial is of degree less than n, then A^{n-1} is a linear combination of $I_n, A, \dots, A^{n-2}$, and there can be no cyclic vector. This shows that the minimal polynomial is of degree n and equals the characteristic polynomial $p(x)$. The surjective ring homomorphism $GF(2)[x] \to GF(2)[A]$ maps a polynomial $f(x)$ to $f(A)$. By definition of the minimal polynomial, $f(A)$ is the zero matrix if and only if $f(x)$ is a multiple of $p(x)$. This completes the proof. □

In the following, we assume that $c_n = 1$ so that the LFSR and the matrix A are *nonsingular*. This assumption is reasonable, since one could otherwise omit the last register bit and obtain an LFSR of lower degree that generates essentially the same output.

The following Proposition relates the *period* of a linear recurring sequence to the *order* of the associated matrix.

Proposition 6.12. *Let A be the matrix associated to a nonsingular LFSR of degree n with characteristic polynomial $p(x)$ and let* $\operatorname{ord}(A)$ *be the order of A in the multiplicative group of invertible matrices over $GF(2)$, i.e., the smallest exponent $N \geq 1$ such that $A^N = I_n$. Then:*

(1) *The period of any output sequence divides* $\operatorname{ord}(A)$.

(2) *If the initial state is $st = (1, 0, \ldots, 0)^T$, then the period of the associated output sequence is equal to* $\operatorname{ord}(A)$.

(3) *If $p(x)$ is irreducible, then the period of any nonzero sequence equals* $\operatorname{ord}(A)$.

Proof. Let st be an initial state (viewed as a column vector). Then the sequence of subsequent states is

$$st,\ A \cdot st,\ A^2 st,\ A^3 st,\ \ldots.$$

Let m be the least period of that sequence. Since $A^{\operatorname{ord}(A)} = I_n$ and hence $A^{\operatorname{ord}(A)} st = st$, we see that $m \mid \operatorname{ord}(A)$, which proves (1). If $st, A \cdot st, \ldots, A^{n-1} st$ form a basis of $GF(2)^n$, then the period of the output sequence and the period of A in the group of invertible matrices coincide. For $st = (1, 0, \ldots, 0)^T$, one can easily check that the vectors $st, A \cdot st, \ldots, A^{n-1} st$ are linearly independent, and (2) is proved. If $p(x)$ is irreducible, then Propositions 4.67 and 6.11 (6) imply an isomorphism of *fields*

$$GF(2)[x]/(p(x)) \cong GF(2)[A].$$

Therefore, any non-trivial linear combination of the matrices $I_n, A, \ldots, A^{n-1}$ is invertible, which shows that the vectors $st, A \cdot st, \ldots, A^{n-1} st$ are linearly independent for any nonzero state st. This proves (3). □

The above Proposition shows that the maximal period of a nonsingular LFSR with matrix A is $\operatorname{ord}(A)$. Under which conditions is the period equal to the maximal value $2^n - 1$?

Definition 6.13. Let $f(x) \in GF(p)[x]$ be a polynomial of degree $n \geq 1$ with $f(0) \neq 0$. Then $\operatorname{ord}(f)$ (the order of f) is defined to be the smallest integer $N \geq 1$ such that $f(x) \mid x^N - 1$. ◊

It is not difficult to see that the order of f is well defined: consider the quotient ring $GF(p)[x]/(f(x))$ and its group of units $U = (GF(p)[x]/(f(x)))^*$, which has at most $p^n - 1$ elements. If f is irreducible then $\operatorname{ord}(U) = p^n - 1$. Since $f(0) \neq 0$, x is invertible modulo $f(x)$ and Euler's Theorem 4.15 implies that

$$x^{\operatorname{ord}(U)} \equiv 1 \mod f(x).$$

In other words, $f(x)$ divides $x^{\operatorname{ord}(U)} - 1$. In fact, we have

$$\operatorname{ord}(f) = \operatorname{ord}(x) \mid \operatorname{ord}(U),$$

where $\operatorname{ord}(x)$ is the order of x in U.

Proposition 6.14. *Let $p(x) = x^n + c_1 x^{n-1} + \dots + c_{n-1}x + c_n \in GF(2)[x]$ be the characteristic polynomial of a nonsingular LFSR of degree n and suppose $p(x)$ is primitive, i.e.,*

(1) *$p(x)$ is irreducible and*

(2) $\operatorname{ord}(p(x)) = 2^n - 1$.

Then all output sequences with nonzero initial state have maximal period $2^n - 1$ and the LFSR is called maximum-length.

Proof. Let A be the matrix associated to the LFSR and suppose $p(x)$ is primitive. Proposition 6.11 (6) shows that

$$(GF(2)[x]/(p(x)))^* \cong GF(2)[A]^*,$$

and hence $\operatorname{ord}(x) = \operatorname{ord}(A) = \operatorname{ord}(p(x)) = 2^n - 1$. It remains to prove that the period of all nonzero sequences is maximal. But this follows from Proposition 6.12 (3). ☐

Example 6.15. Let $c(x) = x^4 + x + 1$ be the connection polynomial of an LFSR (see Example 6.9). Then $p(x) = x^4 + x^3 + 1$ is the corresponding characteristic polynomial (see Proposition 6.11). It is easy to check that $p(x)$ is irreducible and we show that $p(x)$ is primitive:

We have $\operatorname{ord}(p(x)) = \operatorname{ord}(x)$, where the latter is the order of x in the group of units of $GF(2)[x]/(x^4 + x^3 + 1)$. Since the order of the group of units is 15 and thus $\operatorname{ord}(x) \mid 15$, the order of x must be either 1, 3, 5 or 15. Obviously, $x \not\equiv 1$ and $x^3 \not\equiv 1$. Since $x^5 \equiv x^3 + x + 1 \mod x^4 + x^3 + 1$ one has $x^5 \not\equiv 1$. Hence $\operatorname{ord}(x) = \operatorname{ord}(p(x)) = 15$. For every nonzero initial state, the LFSR generates a sequence of maximal length.

We can also use SageMath to do the work:

```
sage: R.<x> = PolynomialRing(GF(2))
sage: (x^4+x^3+1).is_primitive()
True
```

We have seen that a maximum-length LFSR of degree n generates an output sequence of length $2^n - 1$. In principle, a synchronous stream cipher can be based on an LFSR: first, the initialization algorithm loads the cipher key and the IV into the initial state of the LFSR. The LFSR is clocked a number of times and the output discarded in order to protect the secret key. The following output bits are then used as a keystream.

Although LFSRs have good statistical properties, the output of a *linear* feedback shift register should not be used as a keystream. Having n successive output bits (for example in a known plaintext attack) suffices to reconstruct the state vector. If the feedback coefficients are known, then all subsequent and previous output bits can be computed. Even if the coefficients are kept secret (in contradiction to Kerkhoff's principle; see Remark 2.3), everything can be reconstructed from $2n$ output bits, as the following proposition shows:

Proposition 6.16. *Consider a nonsingular LFSR of degree n with unknown feedback coefficients. If the characteristic polynomial is irreducible, then the feedback coefficients* $c_1, \dots, c_n$ *can be uniquely determined from* $2n$ *consecutive output bits* $y_1, y_2, \dots, y_{2n}$ *(not all zero) of the LFSR.*

Proof. Let A be the unknown matrix associated to the LFSR. We reconstruct n state vectors of the LFSR from the output bits:

$$st = \begin{pmatrix} y_n \\ \vdots \\ y_1 \end{pmatrix},\ A \cdot st = \begin{pmatrix} y_{n+1} \\ \vdots \\ y_2 \end{pmatrix},\ \dots,\ A^{n-1} st = \begin{pmatrix} y_{2n-1} \\ \vdots \\ y_n \end{pmatrix}.$$

Let $x = (c_1\ c_2\ \dots\ c_n)$ be the first row of A. Then:

$$\begin{aligned} y_{n+1} &= x \cdot st, \\ y_{n+2} &= x \cdot (A \cdot st) \\ &\ \vdots \\ y_{2n} &= x \cdot (A^{n-1} st). \end{aligned}$$

We obtain a linear system of equations $y = Mx$, where the rows of M are formed by the vectors $st, A \cdot st, \dots, A^{n-1}st$. Since $p(x)$ is irreducible, we obtain as in the proof of Proposition 6.12 (3) that st, $A \cdot st$, …, $A^{n-1}st$ are linearly independent. Therefore, the linear system of equations $y = Mx$ has a unique solution, the vector of feedback coefficients. □

Example 6.17. Suppose the following output bits of an LFSR of degree 4 are known (in this order):

$$0, 1, 1, 1, 1, 0, 1, 0.$$

We want to reconstruct the feedback coefficients and a state. The first four bits (in reverse order) give the state vector:

$$st = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The subsequent states are $A \cdot st = (1,1,1,1)^T$, $A^2 st = (0,1,1,1)^T$ and $A^3 st = (1,0,1,1)^T$. This yields four linear equations in the unknown feedback coefficients x_1, x_2, x_3 and x_4. The left side of the equations is given by the last four output bits.

$$\begin{aligned} 1 &= 1x_1 + 1x_2 + 1x_3 + 0x_4 \mod 2, \\ 0 &= 1x_1 + 1x_2 + 1x_3 + 1x_4 \mod 2, \\ 1 &= 0x_1 + 1x_2 + 1x_3 + 1x_4 \mod 2, \\ 0 &= 1x_1 + 0x_2 + 1x_3 + 1x_4 \mod 2. \end{aligned}$$

This 4×4 system of linear equations over $GF(2)$ is regular and the unique solution are the feedback coefficients $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ and $x_4 = 1$. In fact, we have taken eight output bits from the LFSR in Example 6.9.

Remark 6.18. The *Berlekamp-Massey* algorithm (see [**MvOV97**]) finds the shortest LFSR that generates a given finite sequence. The degree of the shortest LFSR is called the *linear complexity* of a sequence. Suppose the characteristic polynomial $p(x)$ of a nonsingular LFSR is irreducible and $\deg(p(x)) = n$. Then each nonzero state produces an output sequence of period $\operatorname{ord}(p(x))$ and linear complexity n. With $2n$ given output bits, the Berlekamp-Massey algorithm can compute the feedback coefficients more efficiently than solving a system of linear equations. ◊

We have seen that LFSRs are very efficient and can have a large period, but make weak stream ciphers. The problem is, of course, the linear structure of LFSRs.

One possible approach is to use *filter generators*. A nonlinear function f is applied to the entire state of an LFSR and defines the keystream:

$$y_j = f(s_{j-1}, s_{j-2}, \ldots, s_{j-n}).$$

The multiplications of state bits (AND) can be used along with additions (XOR).

Furthermore, one can use *combination generators*, which combine several LFSRs. A linear or nonlinear Boolean function takes the output bits of each register as input and combines them into a single keystream bit.

Example 6.19. The stream cipher Trivium [**DCP08**], which belongs to the portfolio of the eSTREAM project, combines three shift registers of degree 93, 84 and 111, respectively. The output of each register is defined by a nonlinear filter function, and the input is the XOR-sum of one feedback bit and the output of another register. The keystream at each clock tick is the XOR-sum of the output bits of the three registers. ◊

Yet another approach is to use *irregular clocking*: several LFSRs are combined and a nonlinear function determines whether or not a register is clocked (shifted to the right). If a register is not clocked, then the previous bit is output again.

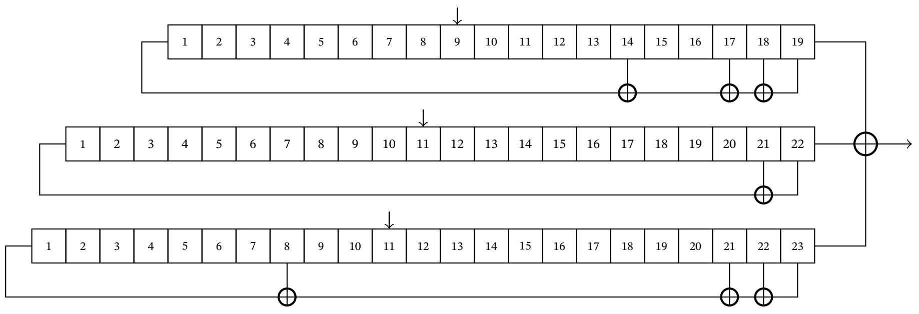


Figure 6.6. The A5/1 cipher combines three LFSRs and uses irregular clocking.

Example 6.20. The A5/1 cipher that is used in GSM mobile networks (2G) combines three LFSRs of degree 19, 22 and 23 (see Table 6.1 and Figure 6.6). In each register, one clocking bit is fixed and a register is clocked, if the clocking bit agrees with the majority of the three clocking bits. Therefore, either all three LFSRs or two of the LFSRs are clocked. The probability that a register is clocked is $\frac{3}{4}$ (see Exercise 5).

Table 6.1. Feedback polynomials and clocking bit of the A5/1 cipher.

LFSR number	Feedback polynomial	Clocking bit (leftmost bit is 1)
1	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	9
2	$x^{22} + x^{21} + 1$	11
3	$x^{23} + x^{22} + x^{21} + x^{8} + 1$	11

Initially, all registers are set to zero. A 64-bit ciphering key (where only 54 bits are secret) and a 22-bit frame number are mixed in. Then the irregular clocking starts: the first 100 output bits are discarded and the next 228 bits are used as the keystream (the first 114 bits for the downlink from the base station to the cellular phone and then 114 bits for the uplink).

The irregular clocking forms the nonlinear component of the cipher. Nevertheless, with current computing resources and large precomputed tables A5/1 is now broken. Better GSM ciphers (A5/3 and A5/4) are available, but whether they are used depends on the network and the mobile phone.

6.3. RC4

The synchronous stream cipher RC4 (*Rivest Cipher* 4) was very popular for many years and was often used to encrypt network traffic (for example in the TLS protocol or for the encryption of Wi-Fi traffic). RC4 is based on permutations of the integers (or bytes) $0, 1, \dots, 255$ and recursively generates output bytes. The *key-scheduling algorithm* (initialization) takes a key (between one and 256 bytes long) and sets up the state array $S[0], S[1], \dots, S[255]$. The *pseudorandom generation algorithm* recursively computes one output byte and updates the state.

RC4 is ideal for software implementations and is very efficient. Unfortunately, the output of RC4 is biased and can be distinguished from a random sequence of bytes.

Algorithm 6.1 RC4 Key Scheduling Algorithm (KSA)

```
Input: Key array $K[0], K[1], \dots, K[n-1]$ of $n$ bytes, $1 \le n \le 255$
Output: State array $S[0], S[1], \dots, S[255]$
1: for $i = 0$ to 255 do
2:     $S[i] = i$
3: end for
4: $j = 0$
5: for $i = 0$ to 255 do
6:     $j = (j + S[i] + K[i \bmod n]) \bmod 256$
7:     Swap the values of $S[i]$ and $S[j]$
8: end for
```

For the remainder of this section, all additions (+) are modulo 256. First, we consider the *key scheduling* (see Algorithm 6.1).

In the first iteration of the **for** loop, one has $i = 0$ and $j = 0 + S[0] + K[0] = K[0]$. The values of $S[0]$ and $S[K[0]]$ are swapped so that $S[0] = K[0]$ and $S[K[0]] = 0$. In the next iteration, one has $i = 1$ and $j = K[0] + S[1] + K[1]$ so that

$$S[1] = K[0] + S[1] + K[1] \mod 256.$$

Hence $S[1]$ is equal to $K[0] + K[1] + 1$, unless $K[0] = 1$ and $S[1] = 0$ as a result of the first iteration. In this case, $S[1] = K[0] + K[1] = 1 + K[1]$.

In the next step, $i = 2$ and (if $K[0] \neq 1$) one gets $j = K[0] + K[1] + 1 + S[2] + K[2]$. Therefore, it is likely that $S[2] = K[0] + K[1] + K[2] + 3$. Note that $S[0]$, $S[1]$ and $S[2]$ may change later in the loop if one of the j-values for $i > 2$ becomes 0, 1 or 2.

By continuing in this fashion, one can show that the most likely value for the i-th state byte after the key scheduling algorithm is

$$S[i] = K[0] + K[1] + \cdots + K[i] + \frac{i(i+1)}{2} \mod 256.$$

For the first nine state values, it can be shown that the above formula holds with more than 30% probability (see [**PM07**]), which is a very significant bias compared to a random permutation.

Algorithm 6.2 RC4 Pseudorandom generation algorithm (PRGA)

```
Input: State array $S[0], S[1], \ldots, S[255]$
Output: Output bytes $B$
1: $i = 0$
2: $j = 0$
3: while Keystream is generated do
4:     $i = i + 1$
5:     $j = (j + S[i]) \bmod 256$
6:     Swap the values of $S[i]$ and $S[j]$
7:     $B = S[(S[i] + S[j]) \bmod 256]$
8:     Output $B$
9: end while
```

Now consider the *pseudorandom generator* (see Algorithm 6.2). The first output byte is $S[S[1] + S[S[1]]]$ and the second byte is

$$S[S[2] + S[S[1] + S[2]]].$$

One can show that the output of RC4 is biased and reveals information about the key. Below, we discuss a famous attack which reveals the key byte $K[3]$.

By construction, RC4 does not use an initialization vector (IV), and thus the keystream must not be re-used with the same key. In practice, the secret key is often re-used and an IV is incorporated into the RC4 key. In the former Wi-Fi encryption standard WEP (*Wired Equivalent Privacy*), a three-byte IV is prepended to the key:

$$K[0] = IV[0],\ K[1] = IV[1],\ K[2] = IV[2].$$

It turned out that this construction is insecure (*Fluhrer, Mantin and Shamir attack* [**FMS01**]): an adversary waits until the first two bytes of the IV are

$$IV[0] = K[0] = 3,\ IV[1] = K[1] = 255.$$

Then the first two iterations of the key scheduling algorithm give

$$S[0] = K[0] = 3,\ S[1] = K[0] + K[1] + 1 = 3 + 255 + 1 \equiv 3 \mod 256.$$

Due to the swapping operations, the first few bytes of the state array are

$$S[0] = 3,\ S[1] = 0,\ S[2] = 2,\ S[3] = 1.$$

The next two iterations yield:

$$S[2] = 3 + 2 + IV[2] = 5 + IV[2] \text{ and}$$
$$S[3] = 5 + IV[2] + 1 + K[3] = 6 + IV[2] + K[3].$$

If we now assume that $S[0] = 3$, $S[1] = 0$ and $S[3] = 6 + IV[2] + K[3]$ are not subsequently modified in the key scheduling algorithm, then the first keystream byte is

$$B = S[S[1] + S[S[1]]] = S[0 + S[0]] = S[3] = 6 + IV[2] + K[3] \mod 256.$$

Since $IV[2]$ is known, the first secret key byte $K[3]$ can be computed from B. In practice, the first plaintext and ciphertext byte and thus the first keystream byte B are often known, for example in Wi-Fi communication.

In the updated RC4-based Wi-Fi security protocol TKIP, the mixing of IV and key was improved, but now TKIP is also deprecated. The RC4 cipher should no longer be used because of serious weaknesses.

6.4. Salsa20 and ChaCha20

The *eSTREAM* project aimed to develop new secure, efficient and compact stream ciphers suitable for widespread adoption. The project ended in 2008, and in 2012 the reviewed eSTREAM portfolio contained seven algorithms in two profiles [**CR12**]:

(1) Profile 1: Stream ciphers with excellent throughput when implemented in software: HC-128, Rabbit, Salsa20/12 and SOSEMANUK.

(2) Profile 2: Stream ciphers which are very efficient - in terms of the physical resources required - when implemented in hardware: Grain v1, MICKEY 2.0 and Trivium.

Profile 1 ciphers use 128-bit keys and profile 2 ciphers 80-bit keys. Extended key lengths are provided by the software ciphers HC-256 and Salsa 20/20 (256-bit keys) and the hardware cipher MICKEY-128 2.0 (128-bit key).

However, it should be noted that the eSTREAM portfolio is not a standardization. The project wants to draw attention to these ciphers and to encourage further cryptanalysis.

In the following, we describe the stream cipher Salsa20/20 (i.e., Salsa20 with 20 rounds and a 256-bit key) [**Ber08b**] and its variant ChaCha20 [**Ber08a**], which has been adopted as a replacement of RC4 in the TLS protocol (see RFC 7905 [**LCM⁺16**]).

Salsa20 is based on three simple operations:

- modular addition of 32-bit words a and $b \mod 2^{32}$, denoted by $a + b$,
- XOR-sum of 32-bit words a and b, denoted by $a \oplus b$,
- circular left shift of a 32-bit word a by t positions, denoted by $a \lll t$.

The Salsa20/20 cipher takes a 256-bit key, a 64-bit nonce and a 64-bit counter. The state array S of Salsa20 is a 4×4 matrix of sixteen 32-bit words. Strings are interpreted in little-endian notation, i.e., the least significant bit of each word is stored first.

$$S = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}.$$

The definition of Salsa20 is based on *quarter-rounds*, *row-rounds* and *column-rounds*. The quarter-rounds operate on four words, the row-rounds transform the four rows and the column-rounds transform the four columns of the state matrix.

Definition 6.21. Let $y = (a,b,c,d)$ be a 4-word sequence. Then a Salsa20 quarter-round updates (a,b,c,d) as follows (see Figure 6.7):

$$\begin{aligned} b &= b \oplus ((a+d) \lll 7), \\ c &= c \oplus ((b+a) \lll 9), \\ d &= d \oplus ((c+b) \lll 13), \\ a &= a \oplus ((d+c) \lll 18). \end{aligned}$$

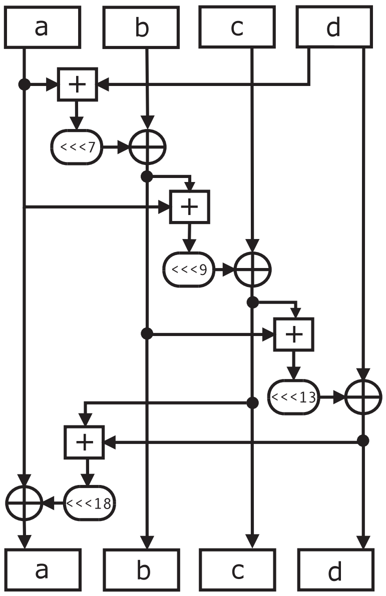


Figure 6.7. A Salsa20 quarter-round.

Example 6.22. (1) *quarter-round* $(0,0,0,0) = (0,0,0,0)$.

(2) We compute $(a,b,c,d) = \textit{quarter-round}\ (0,0,1,0)$, where $1 = \texttt{00 00 00 01}$. Then $b = 0$, $c = 1$, $d = (1 \lll 13) = \texttt{00 00 20 00}$ and $a = (d + c) \lll 18 = (\texttt{00 00 20 01} \lll 18) = \texttt{80 04 00 00}$. The result is

$$(a,b,c,d) = (\texttt{80 04 00 00}, \texttt{00 00 00 01}, \texttt{00 00 00 01}, \texttt{00 00 20 00}). \qquad \Diamond$$

A *row-round* changes the state matrix by applying a quarter-round to each of the rows. The words are permuted before the quarter-round is applied. After the quarter-round operation, the permutation is reversed.

Definition 6.23. Let $S = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$. Then

$$\textit{row-round}\ (S) = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}, \text{ where}$$

$$\begin{aligned}
(z_0, z_1, z_2, z_3) &= \textit{quarter-round}\ (y_0, y_1, y_2, y_3), \\
(z_5, z_6, z_7, z_4) &= \textit{quarter-round}\ (y_5, y_6, y_7, y_4), \\
(z_{10}, z_{11}, z_8, z_9) &= \textit{quarter-round}\ (y_{10}, y_{11}, y_8, y_9), \\
(z_{15}, z_{12}, z_{13}, z_{14}) &= \textit{quarter-round}\ (y_{15}, y_{12}, y_{13}, y_{14}).
\end{aligned}$$

$\Diamond$

The *column-round* function is the transpose of the row-round function: the words in the columns are permuted, the quarter-round map is applied to each of the columns and the permutation is reversed.

Definition 6.24. Let S be a state matrix as above; then

$$\textit{column-round}\ (S) = (\textit{row-round}\ (S^T))^T. \qquad \Diamond$$

A *double-round* is the composition of a column-round and a row-round.

Definition 6.25. Let S be a state matrix as above; then

$$\textit{double-round}\ (S) = \textit{row-round}\,(\textit{column-round}\ (S)). \qquad \Diamond$$

Salsa20 runs 10 successive double-rounds, i.e., 20 quarter-rounds, in order to generate 64 bytes of output. The initial state depends on the key, a nonce and a counter.

Definition 6.26. The Salsa20/20 stream cipher takes a 256-bit key $k = (k_1, \dots, k_8)$ and a unique 64-bit message number $n = (n_1, n_2)$ (nonce) as input. A 64-bit block counter

$b = (b_1, b_2)$ is initially set to zero. The initialization algorithm copies k, n, b and the four 32-bit constants

$$y_0 = \texttt{61707865},\ y_5 = \texttt{3320646E},\ y_{10} = \texttt{79622D32},\ y_{15} = \texttt{6B206574}$$

into the sixteen 32-bit words of the Salsa20 state matrix:

$$S = \begin{pmatrix} y_0 & k_1 & k_2 & k_3 \\ k_4 & y_5 & n_1 & n_2 \\ b_1 & b_2 & y_{10} & k_5 \\ k_6 & k_7 & k_8 & y_{15} \end{pmatrix}.$$

The keystream generator computes the output state by ten double-round iterations and a final addition $\bmod\ 2^{32}$ of the initial state matrix:

$$\text{Salsa20}_k(n, b) = S + \text{double-round}^{10}(S).$$

The block counter c is incremented and the state is newly initialized for additional 64-byte output blocks. The Salsa20 keystream is the serialization of a sequence of 64-byte output blocks:

$$\text{Salsa20}_k(n, 0),\ \text{Salsa20}_k(n, 1),\ \text{Salsa20}_k(n, 2),\ \dots .$$

Remark 6.27. Salsa20 treats strings as *little-endian* integers. For example, if the first four key bytes are 01, 02, 03 and 04, then the corresponding integer is $y_1 = \texttt{04030201}$ in hexadecimal notation. Output words are serialized; the integer 04030201 yields the output bytes 01, 02, 03 and 04 (in this order).

Example 6.28. $\text{Salsa20}_k(n, 0)$ is a zero block if k and n are zero. This should not happen when Salsa20 is used as a stream cipher, since the nonce n must only be used once.

Remark 6.29. Note that the state S is re-initialized for each 64-byte output block and there is no chaining from one block to another. Hence the Salsa20 keystream can be accessed randomly and the computation of 64-byte blocks can be done in parallel. ◊

We turn to the *ChaCha family of ciphers* [**Ber08a**] and describe the ChaCha20 variant described in RFC 8439 [**NL18**]. ChaCha20 is a modification of Salsa20 and we explain the differences to Salsa20.

Definition 6.30. Let $y = (a, b, c, d)$ be a sequence of four 32-bit words. Then a ChaCha *quarter-round* updates (a, b, c, d) as follows:

$$\begin{array}{lllllll} 1) & a = a + b & ; & d = d \oplus a & ; & d \lll 16; \\ 2) & c = c + d & ; & b = b \oplus c & ; & b \lll 12; \\ 3) & a = a + b & ; & d = d \oplus a & ; & d \lll 8; \\ 4) & c = c + d & ; & b = b \oplus c & ; & b \lll 7. \end{array}$$

◊

A ChaCha quarter-round updates each word twice and uses different rotation distances than Salsa20. ChaCha20 also runs ten double-rounds. However, a ChaCha double-round consists of a *column-round* and a *diagonal-round*, which changes words along the main and secondary diagonals.

Definition 6.31. A ChaCha *double-round* is defined by the eight ChaCha quarter-rounds in Table 6.2.

Table 6.2. A column-round and diagonal-round form a ChaCha double-round.

column-round	quarter-round(y_0, y_4, y_8, y_{12})
	quarter-round(y_1, y_5, y_9, y_{13})
	quarter-round(y_2, y_6, y_{10}, y_{14})
	quarter-round(y_3, y_7, y_{11}, y_{15})
diagonal-round	quarter-round(y_0, y_5, y_{10}, y_{15})
	quarter-round(y_1, y_6, y_{11}, y_{12})
	quarter-round(y_2, y_7, y_8, y_{13})
	quarter-round(y_3, y_4, y_9, y_{14})

The RFC version of ChaCha20 described below uses a 12-byte nonce and a 4-byte block counter. The original cipher takes an 8-byte nonce and an 8-byte counter.

Definition 6.32. The ChaCha20 stream cipher takes a 256-bit key $k = (k_1, \dots, k_8)$ and a unique 96-bit message number $n = (n_1, n_2, n_3)$ (nonce) as input. A 32-bit block counter b is initially set to zero. The initialization algorithm copies k, n, b and the four 32-bit constants

$$y_0 = \texttt{61707865},\ y_1 = \texttt{3320646E},\ y_2 = \texttt{79622D32},\ y_3 = \texttt{6B206574}$$

into the sixteen 32-bit words of the ChaCha20 state matrix:

$$S = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ k_1 & k_2 & k_3 & k_4 \\ k_5 & k_6 & k_7 & k_8 \\ b & n_1 & n_2 & n_3 \end{pmatrix}.$$

The ChaCha20 keystream generator works analogously to the Salsa20 generator, but uses ChaCha double-rounds:

$$\text{ChaCha}_k(n, b) = S + \text{double-round}^{10}(S).$$

The block counter b is incremented and the state is newly initialized for each 64-byte output block. The ChaCha20 keystream is the serialization of a sequence of 64-byte output blocks:

$$\text{ChaCha}_k(n, 0),\ \text{ChaCha}_k(n, 1),\ \text{ChaCha}_k(n, 2),\ \dots .$$

Remark 6.33. Salsa20/20 and ChaCha20 are very fast (also in comparison with AES) and encryption requires less than 5 CPU cycles per byte on modern processors.

6.5. Summary

- A stream cipher is a symmetric encryption scheme that uses an internal state and recursively generates keystream. The plaintext is encrypted and the ciphertext is decrypted, respectively, by XORing input words and keystream.
- There are synchronous and self-synchronizing stream ciphers.
- Block ciphers can be turned into stream ciphers by applying the CTR, OFB or CFB mode.
- Linear Feedback Shift Registers (LFSRs) generate output bits by a linear recursion on the state bits. They make weak stream ciphers because of reconstruction attacks, but one can improve the security by using nonlinear filter functions and combinations of several shift registers.
- The stream cipher RC4 was in widespread use over many years, but is now deprecated because of statistical weaknesses.
- Salsa20 and ChaCha20 are examples of promising new stream ciphers.

Exercises

1. Suppose the length of the IV of a synchronous stream cipher is 24 bits. Discuss the security of the cipher.
2. Check whether $p(x) = x^4 + x^3 + x^2 + x + 1 \in GF(2)[x]$ is a primitive polynomial. Suppose $p(x)$ is the characteristic polynomial of an LFSR. Find the periods of output sequences generated by this LFSR.
3. Let $c(x)$ be the connection polynomial of a nonsingular LFSR and let $p(x)$ be the corresponding characteristic polynomial. Show that $p(x)$ is irreducible if and only if $c(x)$ is irreducible. Furthermore, show that $p(x)$ is primitive if and only if $c(x)$ is primitive.
4. Suppose an LFSR of degree 5 is used as a stream cipher and the following plaintext m and ciphertext c is known:
$$m = 00100\ 11000, \quad c = 10110\ 01110.$$
Compute the feedback polynomial, the characteristic polynomial, the period and the complete keystream.
Hint: The first five bits of $m \oplus c$ give a state (reverse the order). The next five bits yield linear equations in the unknown feedback coefficients.
5. Verify that the majority function of three bits x_1, x_2, x_3 is given by
$$X = maj(x_1, x_2, x_3) = (x_1 \wedge x_2) \oplus (x_1 \wedge x_3) \oplus (x_2 \wedge x_3).$$
Show that $Pr[X = x_i] = \frac{3}{4}$ for $i = 1, 2, 3$ if x_1, x_2, x_3 are independent and uniformly distributed.

6. Use SageMath to verify that the feedback polynomials of the A5/1 LFSRs (see Example 6.20) are primitive. Give an upper bound for the period of the A5/1 keystream generator.
7. Suppose an RC4 key satisfies $K[0]+K[1] \equiv 0 \mod 256$. Show that with increased probability the first output byte is $K[2]+3 \mod 256$.
8. Show that the quarter-round operation in Salsa20 is invertible. Give a description of the inverse map.
9. Give an explicit description of the *column-round* operation in Salsa20 using the *quarter-round* map.
10. Apply a Salsa20 quarter-round to $(1,0,0,0)$, $(0,1,0,0)$ and $(0,0,0,1)$, where $1 =$ `00 00 00 01`.
11. Salsa20 can be seen as a map on the vector space $GF(2)^{512}$. Which Salsa20 operations are not $GF(2)$-linear? Explain your answer.
12. In Salsa20 and ChaCha20, the initial state matrix is added to the resulting state matrix after performing ten double-rounds. Why is this final step important for the security of the cipher?
13. Suppose the diagonal rounds in ChaCha20 are omitted. Discuss the consequences of this modification on the security of the cipher.

Chapter 7

Hash Functions

Hash functions form an important cryptographic primitive. They take a message of arbitrary length as input and output a short digest, which should uniquely identify the input data. In Section 7.1, we discuss the main requirements and properties of hash functions. The widely used Merkle-Damgård construction is explained in Section 7.3, while Sections 7.4, 7.5 and 7.6 deal with the SHA-1 hash function, the SHA-2 family and the new standard hash function SHA-3, respectively.

We refer to [**MvOV97**], [**PP10**], [**KL15**] and [**Ber11**] for additional reading on hash functions.

7.1. Definitions and Security Requirements

In general, a *cryptographic hash function* consists of a polynomial-time key generator and a hash algorithm. The key generation algorithm takes a security parameter 1^n as input and outputs a key k. A *keyed* hash function

$$H_k : \{0,1\}^* \to \{0,1\}^{l(n)}$$

takes a key and a binary string as input and outputs a hash value of length $l(n)$. If the input length is restricted to $l'(n) > l(n)$, then H_k is called a *compression function*.

Since hash values are used as *message digests* or unique *identifiers*, their main requirement is *collision resistance*. A collision is given by two input values $x \neq x'$ such that $H(x) = H(x')$.

Definition 7.1. A function $H = H_k : D \to R$, where H, k, D and R depend on a security parameter n, is called *collision resistant* if the probability that a probabilistic polynomial-time adversary finds a collision $H(x) = H(x')$, where $x,\ x' \in D$ and $x \neq x'$, is negligible in n. ◊

If the domain D is larger than the range R, then H cannot be injective and collisions must therefore exist. However, the probability of finding collisions with limited computing resources may be very small.

There are two related requirements which are weaker than collision resistance. We only give informal definitions:

- *Second-preimage resistance* or *weak collision resistance* means that an adversary, who is given a uniform $x \in D$, is not able to find a second preimage $x' \in D$ with $x \neq x'$ such that $H(x) = H(x')$.
- *Preimage resistance* or *one-wayness* means that an adversary, who is given a uniform $y \in R$, is not able to find a preimage $x \in D$ such that $H(x) = y$.

One can show that collision resistance implies second-preimage resistance and preimage resistance (see Exercise 1).

In practice, hash functions are usually *unkeyed* or the key is fixed. Unkeyed hash functions $H : \{0,1\}^* \to \{0,1\}^l$ have a theoretical disadvantage: they are fixed functions and a collision can be found in constant time. However, this can be inaccessible if l is large. Therefore, one requires that it is *computationally infeasible* to produce a collision. In particular, not even a single collision should be known.

Remark 7.2. An ideal unkeyed hash function is called a *random oracle*. The output of a random oracle is uniformly random, unless the same input is queried twice, in which case the oracle returns the same output. One can construct a pseudorandom generator (see Definition 2.32) and a pseudorandom function (Definition 2.39) from a random oracle (see [**KL15**]). However, implementations of a random oracle are impossible: it must have some compact description, and hence the output of any real-world instance is deterministic and not random.

The random oracle model is used in some security proofs, and one hopes that concrete instantiations of hash functions are sufficiently close to that assumption. A security guarantee in the random oracle model can only be relative: a scheme is secure assuming that the hash function has no weaknesses and produces uniform output. $\diamondsuit$

The output length of a hash function should not be too short. In fact, the *Birthday Paradox* shows that collisions occur surprisingly often (see Proposition 1.61):

Proposition 7.3. *Let k be the number of independent samples drawn from a uniform distribution on a set of size N. If $k \approx 1.2\sqrt{N}$, then the probability of a collision is around 50%.* $\diamondsuit$

If we consider hash values of length l and assume a uniform distribution, then collisions occur after hashing around $\sqrt{2^l}$ messages. For example, if $l = 80$ then around 2^{40} hash values are likely to have a collision, and this can be done on a standard PC. In order to minimize the risk of random collisions, hash values should be at least 200 bits long.

The construction of a secure hash function is not easy, and many obvious definitions do not give collision-resistant functions (see Exercises 2 and 3).

7.2. Applications of Hash Functions

Hash functions have many applications. Firstly, the hash value can be used as a short *identifier* of data. The identifier is unique as long as the hash function is collision resistant. Hashes can be used to verify the *integrity* of messages and files if the verifier has access to the authentic message digest. Hash tables can speed up searching. Hashes can also protect passwords: many operating systems and applications store *password hashes* instead of plaintext passwords. When a user logs in with their username and password, the password is hashed and compared to the stored password hash for that user. Often, the password and a *salt value* is hashed in order to impede the use of lookup tables.

Hashes play an important role in *signature schemes*, which are explored in Chapter 11. Furthermore, message authentication codes, pseudorandom functions and key derivation functions are often based on hash functions (see Chapter 8).

Hashes can also serve as an identifier of a *sequence* $x_1, x_2, \dots, x_t$ of messages. However, the obvious approach to compute

$$H(x_1||x_2||\dots||x_t)$$

is not very efficient, since the verification requires *all* data blocks. In the example below, we will see that *Merkle trees* are more efficient at handling a large number of blocks.

Example 7.4. A *blockchain* is a sequence of linked blocks. Each block contains the hash value of the previous block (see Figure 7.1). A blockchain can be used as a *distributed ledger*, which records transactions in an efficient and verifiable way. Transactions in a block cannot be modified without changing all subsequent hash values.

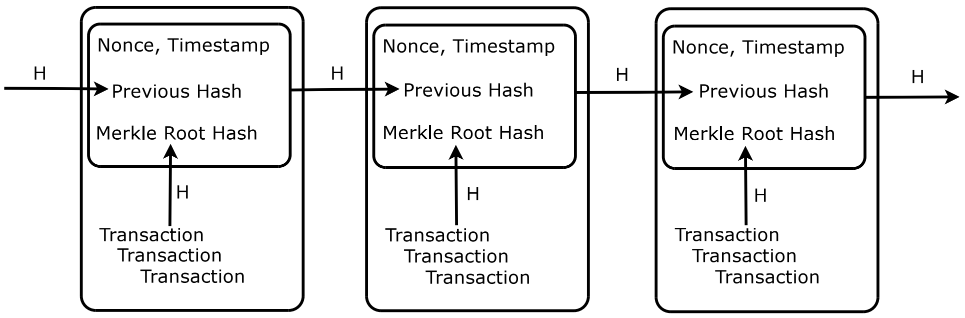


Figure 7.1. Three linked blocks of a blockchain.

The hashes of transactions $T_1,\ T_2,\ T_3,\ \ldots$ form the leaves of a binary tree called the *Merkle tree*. The nodes further up are hashes of two children nodes and the root of the Merkle is the top hash value (see Figure 7.2).

$$\begin{tikzcd}
 & & & H_{root} = H(H_{12}\|H_{34}) & & & \\
 & H_{12} = H(H(T_1)\|H(T_2)) \arrow[urr] & & & & H_{34} = H(H(T_3)\|H(T_4)) \arrow[ull] & \\
H(T_1) \arrow[ur] & & H(T_2) \arrow[ul] & & H(T_3) \arrow[ur] & & H(T_4) \arrow[ul]
\end{tikzcd}$$

Figure 7.2. Merkle tree.

This works with any even number of transactions. The root hash forms an identifier for *all transactions* in a block, and changing a single transaction would completely change the root hash. Individual transactions can be verified by their *hash path* from the leaf to the root.

Suppose we want to prove that a transaction T_3' is included in the blockchain; then we only need to provide the hashes $H_4 = H(T_4)$ and H_{12} along with T_3'. The verifier checks the hash path by computing $H(T_3')$, $H_{34}' = H(H(T_3')\|H_4)$ and $H_{root}' = H(H_{12}\|H_{34}')$. Finally, they verify if H_{root}' coincides with the root hash H_{root} which is stored in the blockchain. This is very efficient, even for larger trees with thousands of leaves, and Merkle trees have many applications beyond blockchains.

Blockchains are used by many *cryptocurrencies*. The blockchain records the transactions of previously unspent cybercoins from one or more input addresses to one or more output addresses. Each new block contains a *proof-of-work*; by adapting the nonce value, a *miner* has to find a hash value of the new block that is smaller than the network's *difficulty target*. This may require a huge number of hashing operations and consume significant computing resources as well as a lot of energy. The miner is rewarded with new cybercoins. The proof of work protects the blockchain against manipulations and complicates forks.

7.3. Merkle-Damgård Construction

A common approach to the construction of hash functions is the *Merkle-Damgård transform*. This construction has found widespread use, including the MD-SHA family. The Merke-Damgård transform is based on a *compression function*, which maps $n + l$ input bits to n output bits. The compression function is applied recursively. First, the message is padded and segmented into blocks of length l. Then the first block is compressed. The output and the next input block are compressed again and this is repeated until the last block is processed. The hash value is defined by the output of the last compression operation (see Figure 7.3).

Definition 7.5. Let $n,\ l \in \mathbb{N}$ and let $f : \{0,1\}^{n+l} \rightarrow \{0,1\}^n$ be a compression function. Let $IV \in \{0,1\}^n$ be an initialization vector. An input message m of arbitrary length is padded by a 1, a sequence of zero bits and the length $L = |m|$, encoded as a 64-bit binary string. The padded message is

$$m' = m||1||0\dots0||L.$$

The number of zeros is chosen such that the length of m' is a multiple of l. We split m' into blocks of length l:

$$m' = m_1||m_2||\dots||m_N.$$

The Merkle-Damgård hash function $H = H_{IV} : \{0,1\}^* \rightarrow \{0,1\}^n$ is defined by recursive application of the compression function f (see Figure 7.3). The last output value defines the hash:

$$\begin{aligned} h_0 &= IV, \\ h_i &= f(h_{i-1},\ m_i) \text{ for } i = 1,\ \dots,\ N, \\ H(m) = H_{IV}(m) &= h_N. \end{aligned}$$

The initialization vector IV can be regarded as a *key*, but in practice, IV is a pre-defined constant. ♢

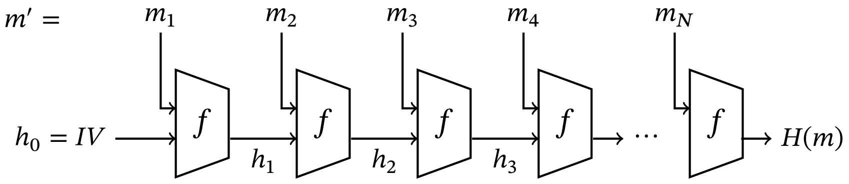


Figure 7.3. Merkle-Damgård construction

The security of a Merkle-Damgård hash function can be reduced to the collision-resistance of the underlying compression function. A collision in a Merkle-Damgård hash function yields a collision in the compression function (see [**KL15**], [**BR05**]).

Theorem 7.6. *If the compression function f is collision-resistant, then so is the associated Merkle-Damgård hash function H.* ♢

The compression function can be based on a block cipher, although this construction is rarely used in practice.

Definition 7.7. (Davies-Meyer) Let E be the encryption function of a block cipher with key length n and block length l. Then a compression function

$$f : \{0,1\}^{n+l} \rightarrow \{0,1\}^l$$

can be defined as follows:

$$f(k,m) = E_k(m) \oplus m.$$ ♢

One can show that this construction defines a collision-resistant compression function in the *ideal cipher model*. A block cipher that is chosen uniformly at random from all block ciphers with n-bit keys and l-bit input/output strings is called an *ideal cipher*. An ideal cipher is a family of *independent* permutations. This is stronger than the standard notion of pseudorandomness and includes protection against *related-key attacks* (see Remark 2.44). Although ideal ciphers cannot be implemented and it is unclear whether real-word block ciphers (for example AES) behave like an ideal cipher, security proofs in the ideal cipher model can still be useful: a scheme can be proven to be secure (for example, the above Davies-Meyer construction), unless an adversary exploits weaknesses of the underlying block cipher.

7.4. SHA-1

Until recently, SHA-1 was a widely used Merkle-Damgård hash function, and in the following we describe its compression function f. As input, the function takes a 160-bit status vector and a 512-bit message block and outputs an updated 160-bit status:

$$f : \{0,1\}^{160+512} \rightarrow \{0,1\}^{160}.$$

As described in the Merkle-Damgård construction, the message is split into 512-bit blocks. The last block is padded by a binary 1, a sequence of zeros and the binary 64-bit representation of the length of the message.

The initial 160-bit status vector is $h_0 = IV = H_1||H_2||H_3||H_4||H_5$, where:

$$\begin{aligned} H_1 &= \texttt{67452301}, \\ H_2 &= \texttt{EFCDAB89}, \\ H_3 &= \texttt{98BADCFE}, \\ H_4 &= \texttt{10325476}, \\ H_5 &= \texttt{C3D2E1F0}. \end{aligned}$$

The compression function $f(h, m)$ is defined by a combination of very efficient linear and nonlinear operations (XOR, bit-rotations, NOT, OR, AND, modular additions), which are applied in 80 consecutive rounds.

A 512-bit message block $m = W_0||W_1|| \dots ||W_{15}$ is subdivided into 16 words of length 32 bits. By XOR operations and a circular left shift by one position, 64 additional words $W_{16}, \dots, W_{79}$ are generated:

$$W_j = (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) \lll 1 \text{ for } 16 \leq j \leq 79.$$

The 160-bit input vector $h = H_1||H_2||H_3||H_4||H_5$ is subdivided into five 32-bit words and copied to the initial status vector:

$$A||B||C||D||E \leftarrow H_1||H_2||H_3||H_4||H_5.$$

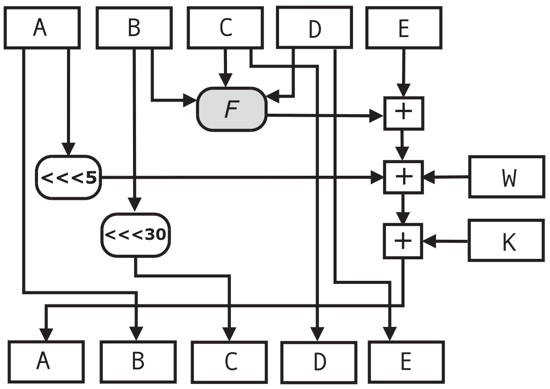


Figure 7.4. One round of the SHA-1 compression function f. The 32-bit status words A, B, C, D, E are updated. In each round, a 32-bit chunk W of the input message is processed. F is a nonlinear bit-function, K a 32-bit constant (both depending on the round number) and $+$ denotes addition modulo 2^{32}.

Then, 80 rounds of the SHA-1 compression function are performed (see Figure 7.4), which update the status words $A||B||C||D||E$. In round j, the 32-bit message word W_j is processed. A bit-function F (defined by AND, OR, NOT and XOR operations) and a constant K are used. The function F and the constant K change every 20 rounds (see Table 7.1).

Table 7.1. Keys and bit functions in SHA-1.

j	K	F		
$0 \leq j \leq 19$	5A827999	$Ch(B,C,D)$	$=$	$(B \wedge C) \oplus (\neg B \wedge D)$
$20 \leq j \leq 39$	6ED9EBA1	$Parity(B,C,D)$	$=$	$B \oplus C \oplus D$
$40 \leq j \leq 59$	8F1BBCDC	$Maj(B,C,D)$	$=$	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$
$60 \leq j \leq 79$	CA62C1D6	$Parity(B,C,D)$	$=$	$B \oplus C \oplus D$

After completing 80 rounds, the compression function outputs

$$f(h,m) = (A + H_1 \mid\mid B + H_2 \mid\mid C + H_3 \mid\mid D + H_4 \mid\mid E + H_5),$$

where $+$ denotes addition modulo 2^{32}.

The *security* of SHA-1 has been intensively studied and it has been anticipated that collisions can be found with available computing resources. In February 2017, a first collision was announced [**SBK⁺17**]. The attack required 2^{63} SHA-1 calls and took approximately 6500 CPU years and 100 GPU years. They chose a prefix P and found two different 1024-bit messages $M^{(1)}$ and $M^{(2)}$ such that

$$H(P||M^{(1)}) = H(P||M^{(2)}).$$

Since P was chosen to be a preamble for PDF documents (see Example 7.8), the collision makes it possible to fabricate two different PDF files with the same SHA-1 hash value, and impressive examples have been published. $M^{(1)}$ and $M^{(2)}$ each consist of two 512-bit blocks. The Merkle-Damgård iteration that takes the first block of $M^{(i)}$ as input produces a *near collision*, and the second block then gives a full collision. Since both messages have the same length, appending the padding data including the length preserves the collision.

Example 7.8. We check the collision found by [**SBK+17**]. First, define the prefix and the messages.

```
sage: prefix='255044462d312e330a25e2e3cfd30a0a0a312030206f626a0a3c3c2f57696474
      68203220302052 2f48656967687420332030205 22f547970652034203020522f53756274
      797065203520302052 2f46696c746572203620302052 2f436f6c6f7253706163652037 20
      3020522f4c656e677468203820302052 2f42697473506572436f6d706f6e656e7420383e
      3e0a73747265616d0affd8fffe00245348412d3120697320646561642121212121852fec
      092339759c39b1a1c63c4c97e1fffe01'
```

Here is the prefix P in ASCII characters:

```
%PDF-1.3.%.......1 0 obj.<</Width 2 0 R/Height 3 0 R/Type 4 0 R/
Subtype 5 0 R/Filter 6 0 R/ColorSpace 7 0 R/Length 8 0 R/BitsPer
Component 8>>.stream......$SHA-1 is dead!!!!!./..#9u.9...<L.....
```

The messages $M^{(1)}$ and $M^{(2)}$ are as follows:

```
sage: m1= '7f46dc93a6b67e013b029aaa1db2560b45ca67d688c7f84b8c4c791fe02b3df614f
      86db1690901c56b45c1530afedfb76038e972722fe7ad728f0e4904e046c230570fe9d41
      398abe12ef5bc942be33542a4802d98b5d70f2a332ec37fac3514e74ddc0f2cc1a874cd0
      c78305a21566461309789606bd0bf3f98cda8044629a1'
sage: m2= '7346dc9166b67e118f029ab621b2560ff9ca67cca8c7f85ba84c79030c2b3de218f
      86db3a90901d5df45c14f26fedfb3dc38e96ac22fe7bd728f0e45bce046d23c570feb141
      398bb552ef5a0a82be331fea48037b8b5d71f0e332edf93ac3500eb4ddc0decc1a864790
      c782c76215660dd309791d06bd0af3f98cda4bc4629b1'
```

We verify the collision:

```
sage: import hashlib
sage: h1 = hashlib.sha1();h1.update((prefix+m1).decode("hex"))
sage: h1.hexdigest()
'f92d74e3874587aaf443d1db961d4e26dde13e9c'
sage: h2 = hashlib.sha1();h2.update((prefix+m2).decode("hex"))
sage: h2.hexdigest()
'f92d74e3874587aaf443d1db961d4e26dde13e9c'
```

Appending any string m yields another collision:

```
sage: m = 'deadbeef'
sage: h1 = hashlib.sha1();h1.update((prefix+m1+m).decode("hex"))
sage: h1.hexdigest()
'811b0fd07f6109f08da740a72b45d5818455b35e'
sage: h2 = hashlib.sha1();h2.update((prefix+m2+m).decode("hex"))
sage: h2.hexdigest()
'811b0fd07f6109f08da740a72b45d5818455b35e'
```

SHA-1 is now deprecated, and it is recommended to use SHA-2 or the new standard hash function SHA-3 (see below).

7.5. SHA-2

The SHA-2 hash functions SHA-224, SHA-256, SHA-384 and SHA-512 are constructed in a similar way to SHA-1, but use an extended internal state of 256 bits (eight 32-bit words) and larger digests. It is assumed that SHA-2 offers better protection against collision-finding attacks, and at the time of this writing SHA-2 is widely used in security protocols and applications. The SHA-2 family is specified in the standard [**FIP15a**].

Since SHA-2 is a Merkle-Damgård hash function, we only need to define the compression function f and the initial status. In the following, we describe the SHA-256 variant.

The compression function f takes as input a 256-bit status vector and a 512-bit message block and outputs an updated 256-bit status:

$$f : \{0,1\}^{256+512} \to \{0,1\}^{256}.$$

The initial 256-bit status vector $h_0 = H_1 || H_2 || H_3 || H_4 || H_5 || H_6 || H_7 || H_8$ is defined by:

$$\begin{aligned} H_1 &= \texttt{6A09E667}, \\ H_2 &= \texttt{BB67AE85}, \\ H_3 &= \texttt{3C6EF372}, \\ H_4 &= \texttt{A54FF53A}, \\ H_5 &= \texttt{510E527F}, \\ H_6 &= \texttt{9B05688C}, \\ H_7 &= \texttt{1F83D9AB}, \\ H_8 &= \texttt{5BE0CD19}. \end{aligned}$$

A 512-bit message block $m = W_0 || W_1 || \dots || W_{15}$ is split into 16 words of length 32 bits. The functions σ_0 and σ_1 transform 32-bit words by a combination of XOR, right-rotate ($\ggg$) and right-shift ($\gg$) operations:

$$\begin{aligned} \sigma_0(w) &= (w \ggg 7) \oplus (w \ggg 18) \oplus (w \gg 3), \\ \sigma_1(w) &= (w \ggg 17) \oplus (w \ggg 19) \oplus (w \gg 10). \end{aligned}$$

Now 48 additional words $W_{16}, \dots, W_{63}$ are generated:

$$W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16} \text{ for } 16 \le j \le 63.$$

The 256-bit input vector $h = H_1 || H_2 || H_3 || H_4 || H_5 || H_6 || H_7 || H_8$ is subdivided into eight 32-bit words and copied to the initial status vector:

$$A || B || C || D || E || F || G || H \leftarrow H_1 || H_2 || H_3 || H_4 || H_5 || H_6 || H_7 || H_8.$$

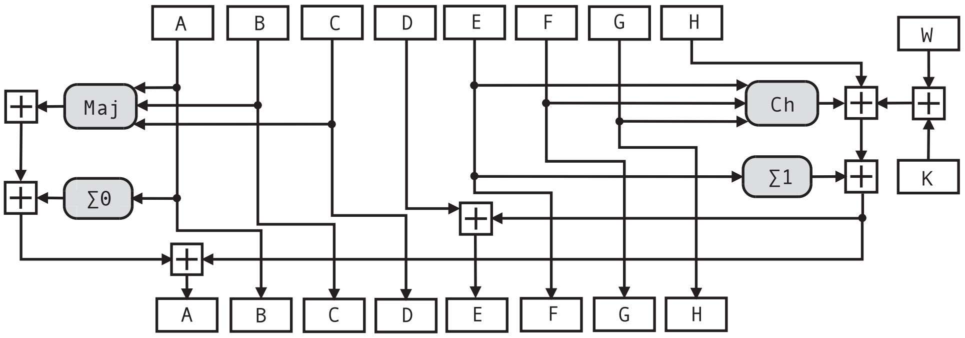


Figure 7.5. One round of the SHA-2 compression function f. The 32-bit status words A, B, C, D, E, F, G, H are updated. In each round, a 32-bit chunk W of the input message is processed. K is a 32-bit constant, which depends on the round number.

Then, 64 rounds of the SHA-2 compression function f are performed (see Figure 7.5), which update the status words $A||B||C||D||E||F||G||H$. In round j, the 32-bit word W_j and the 32-bit constant K_j is processed. The numbers K_j represent the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

In each round, four functions Maj, Ch, Σ_0 and Σ_1 are used which operate on 32-bit words:

$$\begin{aligned} Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), \\ Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z), \\ \Sigma_0(w) &= (w \ggg 2) \oplus (w \ggg 13) \oplus (w \ggg 22), \\ \Sigma_1(w) &= (w \ggg 6) \oplus (w \ggg 11) \oplus (w \ggg 25). \end{aligned}$$

After completing 64 rounds, the compression function outputs

$$f(h,m) = (A + H_1||B + H_2||C + H_3||D + H_4||E + H_5||F + H_6||G + H_7||H + H_8),$$

where $+$ denotes addition modulo 2^{32}.

7.6. SHA-3

Since collisions of MD5 have been found and weaknesses of SHA-1 were already known, in 2007 the American NIST announced a competition to design a new hash function called SHA-3. After narrowing down the candidates in three public rounds, *Keccak* was selected as the winner of the competition in 2012. The main evaluation criteria were *security*, *performance*, *flexibility* and *simplicity* of the design. Keccak is not of Merkle-Damgård type, but rather based on a *sponge* construction (see Figure 7.6). The design and the security claim is explained in [**Ber11**]. The construction is modeled to behave like a *random oracle*.

In 2015, the Keccak variants SHA3-224, SHA3-256, SHA3-384, SHA3-512 with output lengths between 224 and 512 bits were standardized [**FIP15b**]. The SHA-3 instance of Keccak uses a three-dimensional state array of $5 \times 5 \times 64 = 1600$ bits. The unkeyed Keccak-$f[1600]$ permutation operates on the 1600-bit state array and it is assumed that f behaves like a *random permutation*. In each step, $r < 1600$ message bits are processed. r is called the *rate*, and the remaining number of $c = 1600 - r$ bits is called the *capacity*. The Keccak-$f[1600]$ permutation

$$f : \{0,1\}^{1600} \to \{0,1\}^{1600}$$

is parametrized by r and c. SHA-3 specifies the combinations which are shown in Table 7.2.

Table 7.2. Combinations of output length, rate and capacity.

Output length l	Rate r	Capacity c	State length $r + c$
224	1152	448	1600
256	1088	512	1600
384	832	768	1600
512	576	1024	1600

Definition 7.9. The SHA-3 family of hash functions

$$H : \{0,1\}^* \to \{0,1\}^l$$

supports output lengths $l \in \{224, 256, 384, 512\}$. Depending on l, the rate r and the capacity c are fixed (see Table 7.2). First, the input message m is padded such that the length of m' is a multiple of r. The padding rule of the SHA-3 family is to append the pattern 0110 ... 01. The padded message m' is split into blocks $m_1, m_2, \ldots, m_N$ of length r:

$$m' = m||0110\ldots01 = m_1||m_2||\ldots||m_N.$$

The state $s = s_1||s_2$ is initialized by the zero vector $0^r||0^c$. During the *absorbing* phase, the message blocks are XORed into the leftmost r bits of the state and the permutation Keccak-$f[1600]$ is applied to the full state of $r + c = 1600$ bits. The state is updated for each message block:

$$s_1||s_2 \leftarrow f(s_1 \oplus m_i \,||\, s_2) \text{ for } 1 \leq i \leq N.$$

Finally, the SHA-3 hash value is computed using a single *squeezing* operation; $H(m)$ is defined by the leftmost l bits of the resulting state vector s_1 (see Figure 7.6). ◊

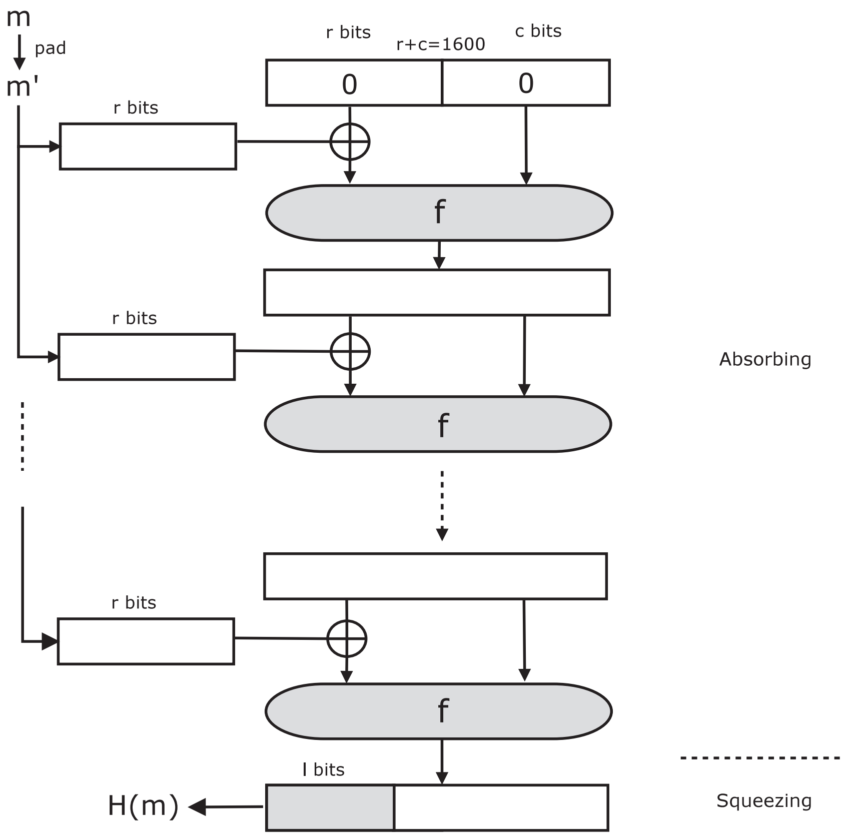


Figure 7.6. Absorbing message blocks of length r into the state and finally squeezing out the SHA-3 hash of length $l < r$.

An advantage of the sponge construction – in comparison to the Merkle-Damgård transform – is that the hash value does *not reveal the full state*, which prevents length extension attacks.

The SHA-3 standard [**FIP15b**] also defines two *extendable-output functions* (XOF) called SHAKE128 and SHAKE256, with which the output can be extended to any desired length. In this case, the Keccak-f function is applied multiple times during the squeezing phase to obtain the required number of output bits.

7.7. Summary

- Hash functions take messages of arbitrary length as input and output a short message digest.
- Hash functions should be collision-resistant; although collisions must exist, it should be very hard to find one.
- Many hash functions (in particular SHA-1 and SHA-2) are based on the Merkle-Damgård construction. A compression function is recursively applied to the input blocks.
- Collisions of SHA-1 have been found using significant computing resources. The SHA-2 variants SHA-224, SHA-256, SHA-384 and SHA-512 are constructed in a similar way to SHA-1, but they generate longer digests and are assumed to be more secure.
- SHA-3 is the new standard hash function. SHA-3 is not a Merkle-Damgård hash function, but it is based on a sponge construction. The internal state array has 1600 bits and the Keccak-$f[1600]$ permutation operates on the state.

Exercises

1. Consider a hash function H. Explain why collision resistance implies second-preimage resistance and the second-preimage resistance implies preimage resistance.
2. Why are linear or affine hash functions not collision-resistant?
3. Why must the output of a collision-resistant hash function depend on *every* input bit?
4. Assume that a collision-resistant hash function is modified as described below. Is it still a collision-resistant hash function?
   (a) The low bit of the message is set to 1. The resulting message is hashed.
   (b) The low bit of the message is flipped and then the message is hashed.
   (c) All bits are flipped and then the message is hashed.
   (d) The message is split into blocks of a fixed length, the blocks are XORed and the result is hashed.
5. Consider a Merkle-Damgård hash function H. Show that an adversary, who knows a hash value $H(m)$ but not the input m, can generate messages m' and compute the hash $H(m||m')$. This is called a *length extension attack*.
6. Find a reason why in the Merkle-Damgård construction the padded message includes the encoded length.
7. Let E be a block cipher satisfying all necessary security assumptions (pseudorandomness and even an ideal cipher) and consider the compression function

$$f(k,m) = E_k(m).$$

Show that f – in contrast to the Davies-Miller construction – is not collision-resistant.

8. Give a table of values of the Boolean functions Ch and Maj used by SHA-1 and SHA-2. Show that the XOR ($\oplus$) operations in these functions can be replaced by OR ($\vee$).
9. Suppose m is a message of length 10^9 bits. How many calls to the SHA-1 compression function, the SHA-2 compression function and the SHA-3 permutation Keccak-$f[1600]$, respectively, are required to compute the hash value $H(m)$?
10. Explain how the rate r and the capacity c are related to the performance and the security level of SHA-3.
11. Suppose Keccak-$f[1600]$ was a linear map on $GF(2)^{1600}$. Fix a SHA-3 output length. Show that a collision in the associated SHA-3 hash function can be constructed using an efficient algorithm.
12. Suppose Keccak-$f[1600]$ was not a permutation and a collision had been found. Can this be used for a collision in SHA-3 ?

Chapter 8

Message Authentication Codes

A message authentication code (MAC) is a cryptographic tag which protects the *integrity* and the *origin* of a message. A correct tag shows that the data has not been tampered with by an adversary, and it also protects against accidental errors. MACs are widely used (for example in network security protocols), since encryption alone is not sufficient to protect the data. In fact, most encryption schemes cannot prevent the manipulation of messages. Streams ciphers (or blocks ciphers in CTR, OFB and CFB mode) are particularly vulnerable, since an adversary can change selected bits.

Message authentication codes use a *symmetric secret key* for tag generation and verification. This constitutes a major difference to signatures (see Chapter 11), where messages are signed with a private key and verification is performed with a public key. The computation of MACs is usually very fast and they can efficiently protect the integrity of mass data.

We outline the definition of message authentication codes and their security requirements in Section 8.1. Practical constructions of MACs, based on block ciphers in CBC mode and on hash functions, are covered in Sections 8.2 and 8.3, respectively. The combination of encryption and message authentication as well as authenticated encryption schemes are discussed in Section 8.4.

For additional reading, we refer to [**KL15**] and [**GB08**].

8.1. Definitions and Security Requirements

Message authentication codes consist of key generation, tag generation and a verification algorithm.

Definition 8.1. A *message authentication code* is given by the following spaces and polynomial-time algorithms:

- A message space $\mathcal{M}$.
- A key space $\mathcal{K}$.
- A key generation algorithm $Gen(1^n)$ that takes a security parameter 1^n as input and outputs a key k.
- A tag generation algorithm, which may be randomized. It takes a message m and a key k as input and outputs a tag $\mathrm{MAC}_k(m)$.
- A deterministic verification algorithm that takes a key k, a message m and a tag t and outputs 1 if the tag is valid, or otherwise 0.

Canonical verification means to re-compute $\mathrm{MAC}_k(m)$ and to output 1 if $t = \mathrm{MAC}_k(m)$, and 0 otherwise. Canonical verification is only possible if the tag generation is *deterministic*. ◊

A message authentication tag is usually short and does not include the message. Verification therefore requires the message, the tag and the key. Note that hash values can also be leveraged to verify the integrity of data. However, the verifier needs to access the authentic hash value, which is impossible in many applications.

The security of message authentication codes is determined by the difficulty to *forge a valid tag* without knowing the key. We assume that an adversary can choose messages and obtain a valid MAC. This is called a *chosen message attack* and corresponds to a situation in practice, where many messages and their MACs are known. The scheme is considered to be insecure if an adversary can generate a new message and an associated valid MAC in polynomial time.

Definition 8.2. Suppose a message authentication code is given. Consider the following experiment (see Figure 8.1): a challenger takes 1^n as input and generates a key $k \overset{\$}{\leftarrow} Gen(1^n)$. An adversary A is given 1^n. They can choose multiple messages m and obtain the tags $\mathrm{MAC}_k(m)$ from an oracle. The adversary succeeds if they can produce a message m, which they did not query previously, and a valid tag t of that message. In this case, the challenger outputs 1, and otherwise 0.

The scheme is called *existentially unforgeable under an adaptive chosen-message attack* (EUF-CMA secure), or just *secure*, if for all probabilistic polynomial-time adversaries, the probability of success is negligible in n.

Remark 8.3. The above experiment can be slightly modified by accepting a valid message/tag pair, where only the tag is new and the message might have been queried before. Unforgeable MACs in this experiment are called *strongly secure*. If canonical verification is used, then secure MACs are automatically strongly secure, since in this case a message uniquely determines the tag.

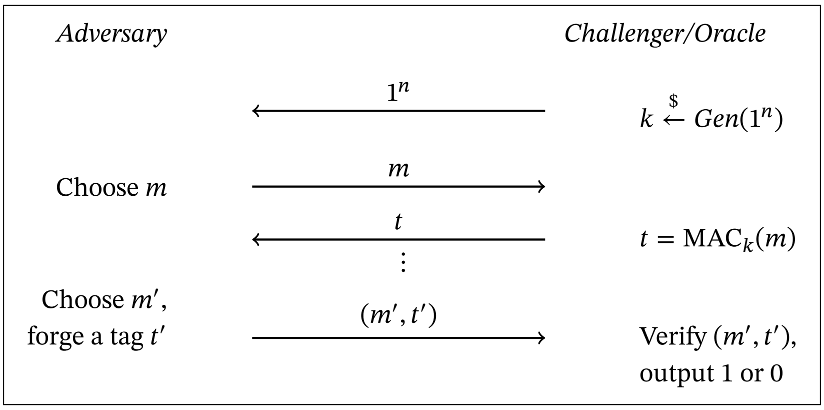

Adversary
Challenger/Oracle
1^n
$k \overset{\$}{\leftarrow} Gen(1^n)$
Choose m
m
t
$t = \mathrm{MAC}_k(m)$
Choose m', forge a tag t'
(m', t')
Verify (m', t'), output 1 or 0

Figure 8.1. MAC forgery experiment.

In the above game, the adversary cannot ask the oracle to verify a tag. Now one can change the experiment by allowing *verification queries*. Since this makes the adversary more powerful, the associated definition of security could be stronger. However, one can show that a strongly secure MAC (for example a MAC with canonical verification) is also secure in this experiment. An adversary can verify a message and a tag by running the original experiment. Since the number of verification queries is polynomial, the security definitions are equivalent.

Remark 8.4. MACs do not protect against the *replay* of messages and tags. If replay protection is required, for example in network protocols, then an additional counter (or a timestamp) should be used. The counter is added to the message and integrated into the tag computation so that the counter cannot be forged. The sender increments the counter for every new message. The receiver keeps track of the counter and discards a message if a counter is re-used or if the tag is invalid. ♢

How can we construct a secure MAC? To begin with, a pseudorandom function (prf) family

$$F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$$

(see Section 2.9) yields a secure MAC.

Theorem 8.5. *Let F be a pseudorandom function family with key length n and block length l. We define an associated MAC: The key generation algorithm $Gen(1^n)$ takes a security parameter 1^n as input and outputs a uniform key $k \overset{\$}{\leftarrow} \{0,1\}^n$. The MAC takes a a message m of length l and a key of length n as input and outputs*

$$\mathrm{MAC}_k(m) = F_k(m).$$

The verification is canonical. This construction defines a secure MAC for messages of length l. ♢

The above Theorem has a proof by reduction (see [**KL15**]). An adversary, who can forge valid MACs, is also able to distinguish F from a random function.

Since the prf-construction only takes messages of fixed length as input, it is rarely used in practice. The construction can be extended to messages of arbitrary length, for example by a sequence of tags (see Exercise 5), but this is not very efficient. In the following two sections, we describe two widely used MAC constructions, CBC MAC and HMAC. They are based on block ciphers and hash functions, respectively.

8.2. CBC MAC

We want to construct a MAC for messages of arbitrary length based on a block cipher. A popular construction is the *CBC MAC* (Cipher Block Chaining Message Authentication Code). The message is encrypted in CBC mode (see Section 2.10) and the *last ciphertext block* is used as an authentication tag.

Definition 8.6. Let $E : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ be a block cipher and $k \overset{\$}{\leftarrow} \{0,1\}^n$ a uniform key. Fix a number N of input blocks and set $\mathcal{M} = \{0,1\}^{Nl}$. The *basic CBC MAC* of a message $m = m_1 || m_2 || \ldots || m_N$ of fixed length Nl is defined by encrypting m in CBC mode and outputting the last ciphertext block:

$$\begin{aligned} c_0 &= 0^l, \\ c_i &= E_k(m_i \oplus c_{i-1}) \text{ for } i = 1, 2, \ldots, N, \\ \mathrm{MAC}_k(m) &= c_N. \end{aligned}$$

The CBC MAC is deterministic, and verification is canonical. $\diamondsuit$

The CBC MAC computation is similar to encryption in CBC mode. However, the initialization vector is a zero string and only the last ciphertext block is output. One can show that the basic CBC MAC is secure for *fixed-length* messages (see [**KL15**]).

Theorem 8.7. *If E is a pseudorandom permutation, then the basic CBC MAC is EUF-CMA secure for messages of fixed length Nl.*

Remark 8.8. The bijectivity of E_k is not required in Definition 8.6, and Theorem 8.7 remains true for a *pseudorandom function family.* $\diamondsuit$

The above basic CBC MAC is not secure for messages of *arbitrary length*: suppose m is a message of length l so that $t = \mathrm{MAC}_k(m) = E_k(m)$. Now an adversary constructs the message $m' = m \,||\, (t \oplus m)$ of length $2l$. Since

$$\mathrm{MAC}_k(m') = E_k(t \oplus (t \oplus m)) = E_k(m) = t,$$

the same tag is also valid for m'. This shows that the basic CBC MAC needs to be modified for messages of variable length. One approach is to prepend the length of the message which prevents this attack (see Exercise 6). Another option is to transform the last input block using a secret key, which prevents the fabrication of valid tags. Below, we describe the CMAC (Cipher-based Message Authentication Code), which is also called OMAC1 (see [**Dwo16**] and RFC 4493 [**SPLI06**]).

The CMAC computation requires the *multiplication* of 128-bit blocks, which is defined in the field

$$GF(2^{128}) = GF(2)[x]/(x^{128} + x^7 + x^2 + x + 1).$$

A 128-bit block $(b_{127}, \dots, b_1, b_0)$ corresponds to the residue class

$$b_{127}x^{127} + \cdots + b_1 x + b_0 \mod x^{128} + x^7 + x^2 + x + 1.$$

Example 8.9. Let $b = (b_{127} \dots b_1\, b_0)$ and $c = (0^{126}10)$. We want to compute $b \cdot c$. The corresponding polynomials are $b_{127}x^{127} + \cdots + b_1 x + b_0$ and x. Their product is $m(x) = b_{127}x^{128} + \cdots + b_1 x^2 + b_0 x$. If $b_{127} = 0$, then the degree of $m(x)$ is less than 128, and therefore $b \cdot c = (b_{126} \dots b_1\, b_0\, 0) = (b \ll 1)$, a left-shift of b by one position. If $b_{127} = 1$, then we have to use the congruence

$$x^{128} \equiv x^7 + x^2 + x + 1.$$

The polynomial $x^7 + x^2 + x + 1$ corresponds to the binary string $R = (0^{120}10000111)$, and thus $b \cdot c = (b_{126} \dots b_1\, b_0\, 0) \oplus R$.

Definition 8.10. Let E be a block cipher with 128-bit block length, k a secret key and $m = m_1 \| m_2 \| \dots \| m_N$ a sequence of 128-bit message blocks, where the last block m_N may be shorter. Then the CMAC of m is defined as the CBC MAC of $m' = m_1 \| m_2 \| \dots \| m_{N-1} \| m'_N$, where the last block is tweaked in the following way: first, two 128-bit subkeys k_1 and k_2 derived:

$$k_1 = E_k(0^{128}) \cdot (0^{126}10) \text{ and } k_2 = E_k(0^{128}) \cdot (0^{125}100).$$

The multiplication of 128-bit blocks is defined in the field $GF(2^{128})$ (see above).

Then set

$$m'_N = \begin{cases} m_N \oplus k_1 & \text{if } |m_N| = 128, \\ (m_N \| 10 \dots 0) \oplus k_2 & \text{if } |m_N| < 128. \end{cases}$$

Hence the subkey k_1 is used if the last block is full length. Otherwise, the last block is padded and the subkey k_2 is used. The verification of the CMAC is canonical. ◊

An adversary cannot produce a valid CMAC without knowing k_1 or k_2, but these subkeys depend on the secret MAC key k. Furthermore, k_1 or k_2 cannot be recovered from valid tags, since m'_N is protected by E_k. One can show that the CMAC construction is secure (see [**IK03**]):

Theorem 8.11. *If E is a pseudorandom permutation or function, then the CMAC construction defines an EUF-CMA secure MAC for messages of variable length.*

Remark 8.12. There are truncated versions of the CMAC, for example AES-CMAC-96 (see RFC 4494 [**SPL06**]), where the tag is defined by the leftmost 96 bits of the CMAC. Of course, the tag must not be too short, say less than 80 bits, since collisions may otherwise be produced and valid MACs forged.

8.3. HMAC

Another widely used MAC construction is based on *hash functions*. Hash functions are usually faster than encryption algorithms. However, hash functions are *unkeyed* in practice, so they cannot be used directly as MACs. But note that the general Merkle-Damgård transform (see Section 7.3) takes an initialization vector (or key) IV as input. The obvious *prefix* construction $H_k(m) = H(k, m)$ (with $k = IV$) or $H_k(m) = H(k||m)$ (for an unkeyed hash function with fixed IV) is *insecure* for messages of variable length if H is a Merkle-Damgård hash function (*length extension attack*; see Exercise 8). Note that the SHA-3 family is not vulnerable to this attack.

The *Hash-based Message Authentication Code* (HMAC) is based on *two nested hashing* operations and protects against length extension attacks. HMAC is described in RFC 2104 [**HK97**] and standardized in [**FIP08**].

Definition 8.13. Let H be a Merkle-Damgård hash function and suppose b is the input block length in bytes of the underlying compression function. For SHA-1 and SHA-256, one has $l = 512$ bits and thus $b = 64$ bytes. The message space is $\mathcal{M} = \{0,1\}^*$ and HMAC keys $k \overset{\$}{\leftarrow} \{0,1\}^n$ are chosen uniformly at random. We assume that the byte length of k is, at most, b. Define `ipad` and `opad` strings by repeating the bytes `36` and `5C`, respectively, b times. The key k is padded by zeros such that the byte length of $\overline{k} = (k \,||\, 0 \ldots 0)$ is b. Then the *HMAC* message authentication tag of a message m is defined as

$$\mathrm{HMAC}(k, m) = H(\overline{k} \oplus \texttt{opad} \,||\, H(\overline{k} \oplus \texttt{ipad} \,||\, m)\,).$$

The verification of a message m and a tag t is canonical: compute $\mathrm{HMAC}(k, m)$ and compare the result with t. The tag is valid if $t = \mathrm{HMAC}_k(m)$. ♢

The HMAC construction is a special case of a *nested MAC* (NMAC). Let H be a keyed hash function and suppose k_1, k_2 are two secret keys. Then define

$$\mathrm{NMAC}_{k_1||k_2}(m) = H_{k_2}(H_{k_1}(m)).$$

k_1 is called the *inner key* and k_2 the *outer key*. One can show that NMAC is a secure MAC if H is based on a pseudorandom compression function ([**Bel06**]):

Theorem 8.14. *Let $f : \{0,1\}^{n+l} \to \{0,1\}^n$ be a compression function that takes a key of length n and a message block of length l as input. Let H be the associated keyed Merkle-Damgård hash function* (*without length padding*)*. The NMAC function defined above takes a key of length $2n$ and a message of arbitrary length as input and outputs a tag of length n. If f is a pseudorandom function for messages of fixed length, then NMAC is a pseudorandom function and a secure MAC for messages of arbitrary length.* ♢

The above security guarantee for NMAC is quite strong. Does it also apply to HMAC? The main differences are: a) HMAC uses an unkeyed hash function and is keyed via the data input, b) length padding is applied, and c) the HMAC keys $\overline{k} \oplus \texttt{opad}$ and $\overline{k} \oplus \texttt{ipad}$ are not independent. Nevertheless, one has the following result [**Bel06**]:

Theorem 8.15. *Let $f : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^n$ be a compression function and let $\overline{f} : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^n$ be the dual function with the same values as f, but keyed via the second component. Let H be the Merkle-Damgård hash function associated with f. If f is a prf and $\overline{f}$ is a prf under restricted related-key attacks, respectively, then HMAC is a pseudorandom function and an EUF-CMA secure MAC for messages of arbitrary length.* $\diamondsuit$

The above Theorem reduces the security of HMAC to the pseudorandomness of f and $\overline{f}$. The related-key attack against $\overline{f}$ can be restricted to two keys ($\overline{k} \oplus \texttt{opad}$ and $\overline{k} \oplus \texttt{ipad}$) and two oracle queries. Note that only pseudorandomness is required, so that HMAC could still be secure when used with hash functions whose collision resistance is compromised.

Remark 8.16. HMAC is widely used in practice, not only as a message authentication code, but also as a pseudorandom function and as a building block in key derivation functions. For example, an *HMAC-based Extract-and-Expand Key Derivation Function* (*HKDF*) is described in RFC 5869 [**KE10**]. Multiple HMAC calls with the same key and different input data can generate the desired number of pseudorandom output bits. $\diamondsuit$

Truncated versions of HMAC are often used for message authentication, for example HMAC-SHA1-80. These variants are defined by the leftmost t bits of the HMAC value. It is recommended that t should not be less than 80.

Remark 8.17. HMAC was designed for Merkle-Damgård hash functions, for example MD5 and SHA-1, which are vulnerable to length extension attacks. SHA-3 (Keccak) does not need the nested approach and a MAC can be defined by prepending the key to the message. The NIST publication [**KCP16**] describes the Keccak Message Authentication Code (KMAC). Keccak can also be used as a pseudorandom function and a key derivation function.

8.4. Authenticated Encryption

In practice, encryption and message authentication are often combined in order to protect the *confidentiality* and the *integrity*. This should also prevent *tampering with ciphertext messages*. We are thus looking for a CCA-secure and *unforgeable* encryption scheme.

Definition 8.18. Suppose an encryption scheme is given. Consider the following experiment (see Figure 8.2): a challenger takes 1^n as input and generates a key $k \overset{\$}{\leftarrow} Gen(1^n)$. An adversary A is given 1^n, can repeatedly choose messages m and obtains the ciphertext $c = \mathcal{E}_k(m)$ from an oracle. The adversary succeeds if they can produce a valid ciphertext c', which they did not previously obtain from the oracle. In this case, the challenger outputs 1 and otherwise 0. The encryption scheme is called *unforgeable* if the probability of success is negligible in n for all probabilistic polynomial-time adversaries. $\diamondsuit$

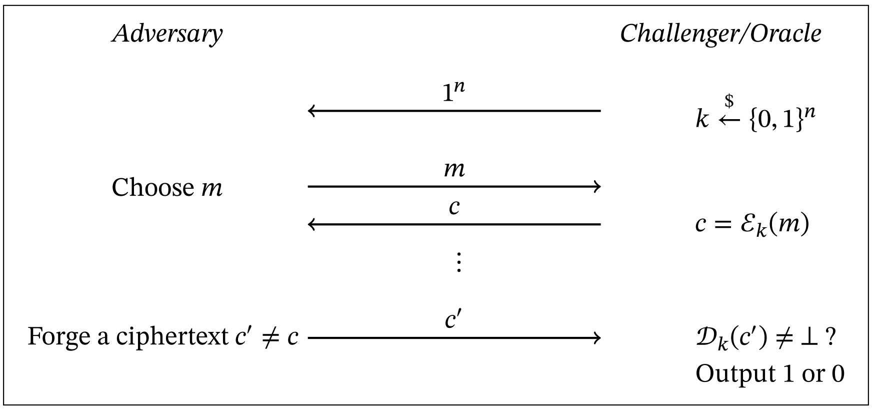

Adversary
Challenger/Oracle
1^n
$k \overset{\$}{\leftarrow} \{0,1\}^n$
Choose m
m
c
$c = \mathcal{E}_k(m)$
$\vdots$
Forge a ciphertext $c' \neq c$
c'
$\mathcal{D}_k(c') \neq \perp$?
Output 1 or 0

Figure 8.2. Ciphertext forgery experiment.

Note that the adversary must produce a *valid* ciphertext c. Depending on the scheme, the decryption of a given string c can be invalid, i.e., $\mathcal{D}_k(c) = \perp$.

Definition 8.19. An encryption scheme is called an *authenticated encryption scheme* if it is CCA2-secure and unforgeable. ◊

We already know (see Remark 2.53) that block ciphers in CBC or CTR mode are malleable and therefore forgeable. An obvious approach to obtaining an authenticated encryption scheme is to combine a CPA-secure encryption scheme and a secure MAC. Several combinations are possible, and it turns out that the *encrypt-then-authenticate* construction is the best choice.

Definition 8.20. Suppose a symmetric-key encryption scheme and a message authentication code is given. We assume that key generation algorithms choose uniform keys of length n. Then define a combined encryption and message authentication scheme as follows: on input 1^n choose two uniform keys $k_E \overset{\$}{\leftarrow} \{0,1\}^n$ and $k_M \overset{\$}{\leftarrow} \{0,1\}^n$. Encryption of a plaintext m with a key $(k_E,\ k_M)$ is defined by

$$\mathcal{E}'_{(k_E,k_M)}(m) = (c,\ t),$$

where $c \overset{\$}{\leftarrow} \mathcal{E}_{k_E}(m)$ and $t = \mathrm{MAC}_{k_M}(c)$. For decryption of $(c,\ t)$, one first verifies the tag and outputs $\mathcal{D}_{k_E}(c)$ if the tag is valid. If the tag is not valid or $\mathcal{D}_{k_E}(c) = \perp$, then output $\perp$. ◊

Note that the tag is computed from the *ciphertext*, not from the plaintext. The next Theorem states that the above definition gives a CCA2-secure and unforgeable scheme if the underlying encryption scheme and the MAC are secure.

Theorem 8.21. *Consider the encrypt-then-authenticate construction defined above. Suppose that the encryption scheme is CPA-secure and the message authentication code is* *a strongly secure MAC (for example a secure MAC with canonical verification). Then the encrypt-then-authenticate construction gives an authenticated encryption scheme.* ♢

Refer to [**KL15**] for a proof. Note that the encrypt-then-authenticate construction turns a CPA-secure encryption scheme into a CCA2-secure scheme.

Remark 8.22. The IP security protocol ESP (Encapsulating Security Payload; see RFC 4303 [**Ken05**]) protects IP packets with the above encrypt-then-authenticate approach. On the other hand, the TLS (Transport Layer Security) 1.2 protocol (see RFC 5246 [**DR08**]), when used with stream ciphers or block ciphers in CBC mode, first computes a message authentication code and then encrypts the plaintext and the MAC. The main problem with this approach is that an error can occur after decryption (padding error) or after tag computation (invalid MAC). If two different error messages are returned or if the computing time is different, then this may leak information about the plaintext. Implementations should therefore only return *one* error message $\perp$ and compute the MAC even if padding is incorrect. Now the TLS protocol 1.3 (see RFC 8446 [**Res18**]) supports only authenticated encryption, for example the GCM mode, which is explained below. ♢

The above encrypt-then-authenticate construction requires two separate algorithms and two independent keys. Now we want to present an operation mode for block ciphers that provides *authenticated encryption* in one pass. The *Galois Counter Mode* (GCM) is an extension of the CTR mode and gained increasing popularity. The GCM mode outputs ciphertext *and* an authentication tag. The tag not only authenticates the plaintext but also *additional authenticated data* (AAD). A variant of GCM is the *message authentication code GMAC*, which only outputs the authentication tag. GCM and GMAC are described in the NIST publication [**Dwo07**].

GCM is defined for block ciphers with 128-bit block length, for example AES. The computation of the tag involves addition (XOR) and *multiplication* of 128-bit blocks. Similar to the CMAC construction (see Section 8.2), the multiplication is defined in the field

$$GF(2^{128}) = GF(2)[x]/(x^{128} + x^7 + x^2 + x + 1).$$

GCM interprets binary strings as polynomials, but – in contrast to CMAC – in *little-endian* order. A 128-bit block $(b_0\, b_1\, \dots\, b_{127})$ thus corresponds to the residue class of the polynomial $b_0 + b_1 x + \cdots + b_{127}x^{127} \bmod x^{128} + x^7 + x^2 + x + 1$.

Example 8.23. The following is similar to Example 8.9, but uses the little-endian convention. Let $b = (b_0\, b_1 \dots b_{127})$ and $c = (010^{126})$. We want to compute $b \cdot c$. The corresponding polynomials are $b_0 + b_1 x + \cdots + b_{127}x^{127}$ and x. Their product is $m(x) = b_0 x + b_1 x^2 + \cdots + b_{127}x^{128}$. If $b_{127} = 0$, then the degree of $m(x)$ is less than 128, and therefore $b \cdot c = (0\, b_0\, b_1 \dots b_{126}) = (b \gg 1)$, a right shift by one position. If $b_{127} = 1$, then we have to use the congruence $x^{128} \equiv x^7 + x^2 + x + 1$. The polynomial $1 + x + x^2 + x^7$ corresponds to $R = (111000010^{120})$, and thus $b \cdot c = (0\, b_0\, b_1 \dots b_{126}) \oplus R$. This computation can be generalized to any factor c (see Exercise 10). ♢

In our description below, we assume that the additional authenticated data (AAD) is 128 bits long at most. AAD may also be empty.

Definition 8.24. (GCM mode) Let E be a block cipher with 128-bit block length. For each encryption, a uniform initialization vector (or a nonce) $IV \overset{\$}{\leftarrow} \{0,1\}^{96}$ is chosen. The plaintext message m is split into blocks of length 128 bits where the last block can be shorter. We write $m = m_1||m_2||\ldots||m_N$. Define a 128-bit counter by $ctr = IV||0^{31}||1$. Applying the CTR mode (see Definition 2.48) gives:

$$c_i = E_k(ctr + i) \oplus m_i \text{ for } i = 1, 2, \ldots, N \text{ and } c = IV||c_1||c_2||\ldots||c_N.$$

Define the 128-bit hash key $H = E_k(0^{128})$ and let $A = AAD$. Then define

$$\begin{aligned} X_1 &= A \cdot H, \\ X_i &= (X_{i-1} \oplus c_{i-1}) \cdot H \text{ for } i = 2, \ldots, N+1, \text{ and} \\ X_{N+2} &= (X_{N+1} \oplus (|A| \mid\mid |c|)) \cdot H. \end{aligned}$$

A and c_N are padded by zeros, if necessary. The multiplication by H is defined in the field $GF(2^{128})$ as described above, and the bit lengths $|A|$ and $|c|$ are represented by 64-bit integers under the big-endian convention. Then the authentication tag t is defined by

$$t = X_{N+2} \oplus E_k(ctr)$$

(see Figure 8.3), and the complete authenticated ciphertext is given by $(c,\ t,\ AAD)$.

For the decryption of $(c,\ t,\ AAD)$, the authentication tag associated to c and AAD is computed using the same formulas as above. If the result is not equal to the given tag t, then output the error symbol $\perp$. Otherwise the plaintext is computed by decrypting c in CTR mode:

$$m_i = E_k(ctr + i) \oplus c_i \text{ for } i = 1, 2, \ldots, N \text{ and } m = \mathcal{D}_k(c) = m_1||m_2||\ldots||m_N. \qquad \Diamond$$

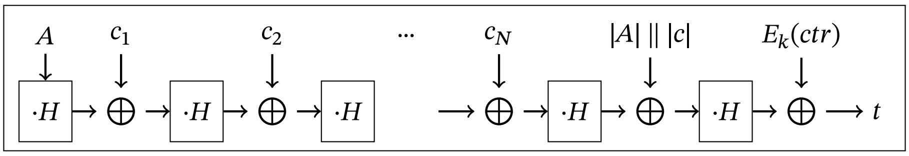


Figure 8.3. Computation of the GCM tag t from the counter mode ciphertext $c = IV||c_1||\ldots||c_N$. $H = E_k(0^{128})$ is the hash key, and the additional authenticated data is denoted by A.

Note that the GCM mode does not strictly follow the encrypt-then-authenticate approach, because the same key and counter is used for encryption and message authentication. The security of GCM is proved in [**MV04**].

8.5. Summary

- A message authentication code (MAC) is a tag that protects the integrity and the authenticity of a message. The computation of a MAC requires a secret key and the message.
- A MAC is secure if it is unforgeable under a chosen message attack.
- The CBC-MAC and CMAC constructions output the last ciphertext block in CBC mode as a tag. CMAC modifies the last the plaintext block before the message is encrypted in CBC mode in order to prevent length extension attacks.
- HMAC is based on a nested hash computation and takes a key and a message as input.
- CMAC and HMAC are secure under certain assumptions.
- Authenticated encryption schemes are CCA2-secure and unforgeable.
- The encrypt-then-authenticate combination of a CPA-secure encryption scheme and a strongly secure MAC gives an authenticated encryption scheme.
- The Galois Counter Mode (GCM) extends the CTR mode and provides encryption as well as message authentication using a single secret key.

Exercises

1. What are possible reasons why a MAC is invalid?
2. Compare the properties of hashes and MACs when used for integrity protection.
3. Show that $\mathrm{MAC}_k(m) = m \oplus k$ defines a secure MAC if k is generated uniformly at random and only used once. Why is this scheme impractical?
4. Suppose a MAC tag does not depend on the first bit of the message. Why is such a MAC insecure under a chosen-message attack?
5. Extend the fixed-length MAC based on a pseudorandom function (see Theorem 8.5) to messages of arbitrary length (see [**KL15**]).
   Tip: Split the message into blocks. A sequence of tags would be a first approach, but this does not protect against re-ordering, truncation and mixing attacks. Tweak the construction in order to prevent such attacks.
6. Show that appending the length L of a message does not turn the basic CBC MAC into a secure MAC for messages of variable length.
   Remark: In contrast, prepending a message with its length is secure.
7. Is it advisable to use the same key for CBC mode encryption and for CBC message authentication?
8. Show that $F_k(m) = H(k \,||\, m)$ is not a secure MAC for Merkle-Damgård hash functions and arbitrary length messages. Why is the HMAC construction not affected by this attack ?

9. Use Sage to verify that the polynomial $x^{128} + x^7 + x^2 + x + 1$ is irreducible in $GF(2)[x]$.
10. Example 8.23 describes how to multiply a 128-bit block b by $c = (010\dots 0)$.
    (a) Give $b \cdot c$ for $c = (0\dots 0)$ and for $c = (10\dots 0)$.
    (b) How can you compute $b \cdot c$ for $c = e_i$, i.e., the i-th bit of c is one and the other bits are zero.
    Hint: $b \cdot X^i = b \cdot X \cdots \cdot X$.
    (c) Now describe the computation of $b \cdot c$ for a general 128-bit block c.
    Hint: $b \cdot (c \oplus c') = (b \cdot c) \oplus (b \cdot c')$.
11. Describe the computation of the GCM tag if the plaintext consists of one 128-bit block and AAD is empty.
12. A block cipher in CTR mode can only achieve CPA security. Describe a chosen ciphertext attack against the CTR mode and show that this attack is not possible if the CTR mode is combined with a secure MAC.

Chapter 9

Public-Key Encryption and the RSA Cryptosystem

This chapter deals with public-key encryption schemes and the RSA cryptosystem. Section 9.1 introduces public-key encryption schemes and defines their security requirements. Section 9.2 explains the widely used RSA encryption algorithm. The security of RSA and the necessary assumptions are covered in Section 9.3. RSA (and other cryptographic schemes) require large prime numbers and Section 9.4 deals with the generation of such primes. The efficiency of RSA and possible optimizations are discussed in Section 9.5. We will see that there are some pitfalls in the application of RSA. This leads to a randomized and padded version of RSA, which is explained in Section 9.6. The security of RSA is closely related to the factoring assumption, and Section 9.7 outlines different factoring algorithms and their complexity.

RSA is a major public-key scheme and is dealt with in all cryptography textbooks, for example in [**PP10**]. For the provable security approach we refer to [**KL15**], [**BR05**], [**GB08**], [**Gol01**].

9.1. Public-Key Cryptosystems

One of the main principles of symmetric encryption is that the encryption and decryption functions are closely related and both use a secret key. For a long time, the pre-establishment of secret keys was assumed to be a prerequisite for cryptosystems. Obviously, a major problem of this type of cryptography is the exchange of secret keys over potentially insecure channels. Furthermore, separate secret keys are needed for each pair of users, so that $\binom{n}{2} = \frac{n(n-1)}{2}$ keys are required in a network with n nodes.

Historically, the idea of using *public encryption keys* is relatively new. A major advantage of this approach is that public keys can be openly exchanged. In addition, one key pair suffices to receive messages from many communication partners. Obviously, it is crucial that an adversary is not able to derive the private decryption key from the public encryption key. Furthermore, the *authenticity* of public keys can represent a problem.

Diffie and Hellman, influenced by Merkle's work, were the first to publish a paper [**DH76**] on public-key cryptography in 1976. They invented a mechanism for secure key distribution over an insecure channel (see Chapter 10). Furthermore, they described the fundamentals of public-key cryptography. Rivest, Shamir and Adleman then published the first public-key encryption scheme (RSA) in 1978 [**RSA78**]. Although Ellis, Cocks and Williamson had already invented public key mechanisms several years before, they were not allowed to publish their results because they worked for the British secret service.

Definition 9.1. (compare Definition 2.1) A *public-key encryption scheme* (public-key cryptosystem) is given by:

- A plaintext space $\mathcal{M}$.
- A ciphertext space $\mathcal{C}$.
- A key space $\mathcal{K} = \mathcal{K}_{pk} \times \mathcal{K}_{sk}$ (pairs of public and private keys).
- A randomized key generation algorithm $Gen(1^n)$ that takes a security parameter n as input and outputs a pair of keys (pk, sk).
- An encryption algorithm $\mathcal{E} = \{\mathcal{E}_{pk} \mid pk \in \mathcal{K}_{pk}\}$ which may be randomized. It takes a public key and a plaintext as input, and outputs the ciphertext or an error symbol $\perp$ if the plaintext is invalid.
- A deterministic decryption algorithm $\mathcal{D} = \{\mathcal{D}_{sk} \mid sk \in \mathcal{K}_{sk}\}$ that takes a private key and a ciphertext as input and outputs the plaintext or an error symbol $\perp$ if the input is invalid.

All algorithms must run in polynomial time. The scheme provides correct decryption if $\mathcal{D}_{sk}(\mathcal{E}_{pk}(m)) = m$ for each key pair $(pk, sk) \in \mathcal{K}$ and all plaintexts $m \in \mathcal{M}$ (see Figure 9.1). ♢

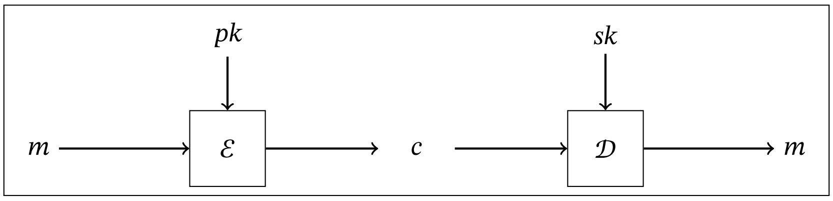


Figure 9.1. Encryption uses a public key *pk* and decryption a private key *sk*.

Now we want to define a public-key counterpart of eavesdropping and CPA-security. We can assume that the public key is known so that an adversary is able to encrypt any chosen plaintext, even without asking an encryption oracle. This implies that security against eavesdropping (EAV) and chosen plaintext attacks (CPA) are equivalent for public-key schemes. The following Definition 9.2 is similar to the previous Definitions 2.24 and 2.25 in the secret-key case.

Definition 9.2. Suppose a public-key encryption scheme is given. Consider the following experiment (see Figure 9.2): a challenger takes the security parameter 1^n as input, generates a key pair $(pk, sk) \in \mathcal{K}$ by running $Gen(1^n)$ and chooses a random bit $b \overset{\$}{\leftarrow} \{0,1\}$. An adversary A is given the public key pk and 1^n. The private key sk and b are not known to the adversary. They can encrypt arbitrary messages using the public key pk. The adversary chooses two messages $m_0,\ m_1 \in \mathcal{M}$ of the same length. Then the challenger encrypts one of the messages, and the ciphertext $c = \mathcal{E}_{pk}(m_b)$ is given to A. The adversary tries to guess b and outputs a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The IND-CPA advantage of the adversary A is defined as

$$\mathrm{Adv}^{\mathrm{ind-cpa}}(A) = |Pr[b' = b] - Pr[b' \neq b]|.$$

The scheme has *indistinguishable encryptions under a chosen plaintext attack* (IND-CPA secure or CPA-secure) if for every probabilistic polynomial time adversary A, the advantage $\mathrm{Adv}^{\mathrm{ind-cpa}}(A)$ is negligible in n. ◊

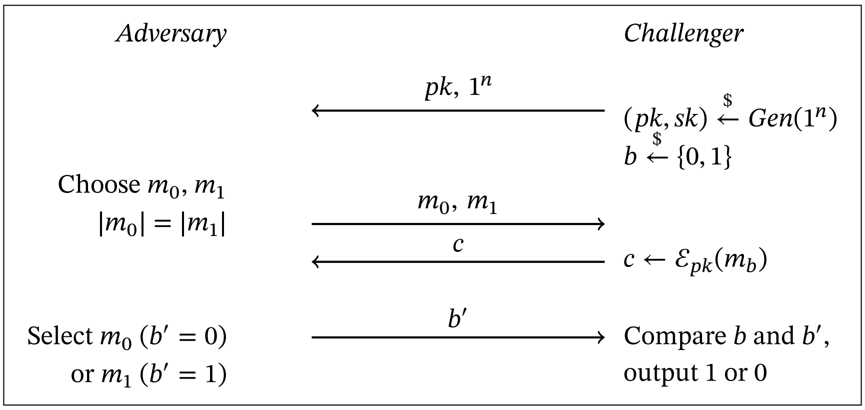


Figure 9.2. Public-key EAV and CPA experiment. The adversary can also encrypt any chosen plaintext.

Since an adversary can encrypt m_0 and m_1 and compare the result with the challenge ciphertext c, it is obvious that a public-key scheme with *deterministic* encryption cannot be IND-CPA secure.

Remark 9.3. A more powerful adversary is able to perform an *adaptive chosen ciphertext attack* (CCA2). In the CCA2 experiment, the adversary can additionally request the *decryption* of arbitrary ciphertexts (before and after choosing two plaintext messages), except that the challenge ciphertext c cannot be queried (compare Figure 2.5 in the secret-key case). $\Diamond$

The construction of secure public-key encryption schemes is far from trivial, since encryption is public but decryption must be hard without the private key. The construction can be based on a family of *trapdoor one-way permutations*. Such permutations are *one-way*, i.e., easy to compute, but *hard to invert* without a *trapdoor* information, which corresponds to the private key. It should be mentioned that hardness of inversion is only required for *uniform* random input. Obviously, an adversary can prepare a list of input values, compute the associated output and use that list for inversion. We refer to [**Gol01**] and [**KL15**] on how to construct a secure public-key encryption scheme from a family of trapdoor permutations.

All known constructions of public-key schemes are based on *hard number-theoretic problems*, which also provide a *security guarantee* for these schemes. This represents an advantage over secret-key schemes, where such guarantees do not exist. However, public-key schemes are much *less efficient*, and, in practice, such schemes are only applied to a few blocks.

The RSA algorithm, which is explained in the next section, uses exponentiation modulo a public composite number N as its one-way permutation. The prime factors of N represent the private trapdoor information that permit the inversion.

9.2. Plain RSA

The plain RSA encryption scheme (schoolbook RSA) requires two large prime numbers; one takes a security parameter 1^n as input, chooses two random primes p and q of size n and sets $N = pq$. By the Chinese Remainder Theorem 4.26,

$$\mathbb{Z}_N \cong \mathbb{Z}_p \times \mathbb{Z}_q \text{ and } \mathbb{Z}_N^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_q^*$$

for $p \neq q$ and $\operatorname{ord}(\mathbb{Z}_N^*) = \varphi(N) = (p-1)(q-1)$. One chooses $e \in \mathbb{Z}$ such that

$$1 < e < (p-1)(q-1) \text{ and } \gcd(e,(p-1)(q-1)) = 1,$$

and computes a modular inverse $d \in \mathbb{Z}$ with

$$1 < d < (p-1)(q-1) \text{ and } ed \equiv 1 \mod (p-1)(q-1).$$

Then $pk = (e, N)$ forms the public key and $sk = (d, N)$ the private key. N is called the *RSA modulus*, e is the *encryption exponent* and d is the *decryption exponent*. The factors p, q and $\varphi(N)$ must remain secret, since d can be efficiently derived from any of these numbers. For encryption, the plaintext is raised to the power e and reduced modulo N. Decryption works similarly, but raises the ciphertext to the power d.

Definition 9.4. The plain RSA encryption scheme is defined by:

- A polynomial-time key generation algorithm $Gen(1^n)$ that takes the security parameter 1^n as input, generates two random n-bit primes p and q, and sets $N = pq$. Furthermore, two integers e and d with
$$ed \equiv 1 \mod (p-1)(q-1)$$
are chosen as explained above. $Gen(1^n)$ outputs the public key $pk = (e, N)$ and the private key $sk = (d, N)$.
- The plaintext and the ciphertext space is $\mathbb{Z}_N^*$.
- The deterministic encryption algorithm takes a plaintext $m \in \mathbb{Z}_N^*$ and the public key pk as input and outputs
$$c = \mathcal{E}_{pk}(m) = m^e \mod N.$$
- The decryption algorithm takes a ciphertext $c \in \mathbb{Z}_N^*$ and the private key sk as input and outputs
$$m = \mathcal{D}_{sk}(c) = c^d \mod N.$$

The scheme is only defined for messages of fixed length. $\Diamond$

For the correctness of the RSA scheme one has to show that
$$(m^e)^d \equiv m \mod N$$
for all $m \in \mathbb{Z}_N^*$. But this follows from Euler's Theorem (see Theorem 4.15 and Exercise 4.4): let $m \in \mathbb{Z}_N^*$; then we have
$$m^{\varphi(N)} \equiv 1 \mod N.$$
Since $ed = 1 + k\varphi(N)$ for some $k \in \mathbb{Z}$, we obtain $m^{ed} = m(m^{\varphi(N)})^k \equiv m \mod N$.

Example 9.5. Suppose Bob's RSA key is given by $p = 29$, $q = 23$, $N = pq = 667$, $e = 3$, $d = 411$. This defines an admissible RSA cryptosystem, since $e = 3$ is relatively prime to $\varphi(N) = (p-1)(q-1) = 616$. The multiplicative inverse $d = 411$ can be computed using the Extended Euclidean Algorithm:
$$616 : 3 = 205 \text{ remainder } 1 \implies 616 = 3 \cdot 205 + 1 \implies 1 = 616 - 205 \cdot 3.$$

Hence $d = (3 \mod 616)^{-1} \equiv -205 \equiv 411$. The public key $pk = (3, 667)$ is published by Bob. If Alice wants to send him the message $m = 108$, she will encrypt it as follows:

$$c = E_e(m) = m^e = 108^3 \mod 667 \equiv 416.$$

She sends $c = 416$ to Bob, who is able to decrypt the ciphertext using his private key $sk = (411, 667)$:

$$m = D_d(c) = c^d = 416^{411} \mod 667 \equiv 108. \qquad \diamondsuit$$

9.3. RSA Security

It is obvious that the security of RSA relies on the hardness of factoring. If an adversary can compute the factors p and q, then they know $\varphi(N) = (p-1)(q-1)$ and easily find the private exponent $d \equiv (e \mod \varphi(N))^{-1}$. Furthermore, the factors p and q can be computed from $\varphi(N)$:

$$\varphi(N) = (p-1)\left(\frac{N}{p} - 1\right).$$

Multiplication by p yields a quadratic equation in p:

$$p^2 - (N - \varphi(N) + 1)p + N = 0.$$

The roots of the quadratic equation $x^2 + (N - \varphi(N) + 1)x + N = 0$ are p and q. Hence $\varphi(N)$ must also be kept secret.

Example 9.6. Consider Example 9.5 and suppose $N = 667$ and $\varphi(N) = 616$ are known. Then the roots of the equation

$$x^2 - (667 - 616 + 1)x + 667 = x^2 - 52x + 667 = 0$$

give $x_1 = p = 29$ and $x_2 = q = 23$. $\diamondsuit$

The factoring of integer numbers has been intensively studied and it is generally assumed that finding large prime factors of a composite number is a *hard problem*.

Suppose a modulus generation algorithm takes the security parameter 1^n as input and outputs two primes p, q. Let $N = pq$. A probabilistic polynomial time adversary A is given N and has to find the factors p and q. Now the *factoring assumption* states that the modulus can be efficiently generated in such a way that an adversary has only a negligible chance of finding the correct factors p and q.

Since no polynomial-time factoring algorithms have been found so far, this assumption is generally believed to be true and forms the basis of major cryptographic schemes. Factoring algorithms are discussed in Section 9.7.

Example 9.7. Suppose the primes p and q are chosen such that the difference $p - q$ is small. In this case, factoring $N = pq$ is not hard, even if p and q are large prime numbers (see Fermat's factorization method in Section 9.7). In fact, the primes should be chosen independently. $\Diamond$

Factoring N breaks RSA, but the opposite statement is not necessarily true. The security of RSA is in fact based on the *RSA assumption*, which states that encryption is a one-way permutation. However, the RSA assumption is stronger than the factoring assumption, since an adversary might attack RSA without factoring the modulus.

Definition 9.8. Consider the following RSA experiment: run the RSA key generation algorithm $Gen(1^n)$ on input 1^n to obtain the parameters p, q, e, d and N. A uniform ciphertext $c \overset{\$}{\leftarrow} \mathbb{Z}_N^*$ is chosen and an adversary obtains 1^n, e, N and c. The adversary has to find $m \in \mathbb{Z}_N^*$ such that $m^e \bmod N \equiv c$. The RSA problem is hard relative to Gen, if for every probabilistic polynomial-time adversary, the probability of finding the correct plaintext m is negligible in n.

The *RSA assumption* states that there is a key generation algorithm such that the RSA problem is hard. $\Diamond$

The RSA assumption means that it is hard to recover the plaintext from a randomly chosen ciphertext, but this does not imply the security of the plain RSA scheme. In fact, the plain RSA encryption scheme is *deterministic* and thus cannot be CPA-secure. This is critical in situations where the possible plaintexts are known or the number of plaintexts is small. Then an adversary can easily find the plaintext simply by encrypting the plaintext candidates. But if the plaintext messages are chosen uniformly at random from a large space, then one might expect that the scheme is secure under the RSA assumption. However, there are a number of pitfalls, which are discussed below. Further details can be found in the survey article [**Bon99**].

(1) Encryption of a short plaintext message m with a small encryption exponent e is insecure. If $c = m^e < N$, then c is computed without modular reduction, and hence the plaintext m can be recovered by computing the *real* e-th root $c^{1/e}$. If $e = 3$ then this *low-exponent attack* can be applied to all messages of length $< \frac{n}{3}$, where $n = \text{size}(N)$. In practice, one often chooses the public exponent $e = 2^{16}+1$, which is large enough to prevent this attack.

(2) If a fixed message m (not necessarily short) is encrypted for e recipients with different RSA moduli, then the Chinese Remainder Theorem allows m to be recovered by computing a real e-th root (*Hastad's broadcast attack*; see Exercise 7).

(3) The modulus N must not be shared among different users, even if individual exponents e and d are used. They can factorize N and therefore compute the private exponents of all users who share this modulus. Furthermore, one can show that sharing the modulus is insecure, even if the users trust each other.

(4) The prime factors of the modulus must not be re-used. If $N_1 = pq_1$ and $N_2 = pq_2$, then $p = \gcd(N_1, N_2)$ can be efficiently computed.

(5) It was shown that small decryption exponents $d < \frac{1}{3}N^{1/4}$ can be efficiently recovered (*Wiener attack*). This attack can be improved to $d < N^{0.292}$. Such d should therefore be avoided. However, if the public exponent e is chosen first, then the probability that d satisfies this condition is very small.

(6) The plaintext of two related messages m_1 and m_2, for example

$$m_2 = am_1 + b \mod N,$$

can be recovered from their ciphertexts c_1 and c_2 if a and b are known and the public exponent e is small (*Franklin-Reiter attack*).

(7) The unknown part of a partially known plaintext can be recovered from the ciphertext if the encryption exponent e is small (*Coppersmith attack*).

(8) The private exponent d can be reconstructed if the least significant $\lceil \frac{\text{size}(N)}{4} \rceil$ bits of d are known (*partial key-exposure attack*).

Furthermore, plain RSA does not provide protection against *ciphertext manipulations* and *chosen ciphertext attacks*:

(1) Plain RSA encryption is *malleable* and the ciphertext can be easily manipulated. If an adversary replaces the ciphertext $c = m^e \mod N$ with $s^e c \mod N$, then the corresponding plaintext becomes $sm \mod N$. Similarly, the ciphertext of a product of plaintexts is congruent to the product of ciphertexts $\mod N$; plain RSA encryption is a *multiplicative* homomorphism.

(2) A *chosen ciphertext attack* against plain RSA is easily performed: if an adversary is given the challenge ciphertext c, then they may ask for the decryption of an unsuspicious-looking ciphertext $c' = cs^e \mod N$, where $s \in \mathbb{Z}_N^*$ and $s \not\equiv 1$. If m and m' are the plaintexts corresponding to c and c', then $m' = ms \mod N$. Hence an adversary can easily compute the plaintext $m = m's^{-1} \mod N$ (see Exercise 6(c)).

9.4. Generation of Primes

RSA and other public key algorithms require large *random prime numbers*. The Prime Number Theorem 3.5 tells us that the asymptotic density of primes among the first N integer numbers is $\frac{1}{\ln(N)}$. Fortunately, the density only decreases logarithmically.

Example 9.9. The density is small, but not too small, since $\ln(x)$ increases slowly. For example, the density of primes among *odd* random numbers less than 2^{2048} is approximately $\frac{2}{2048\ln(2)} \approx 0.0014$. The expected number of trials is $\frac{1}{0.0014} \approx 710$. $\lozenge$

To generate a large prime, choose an *odd random number* of the required size and test its primality. Usually, this requires testing of several hundred candidates, as in Example 9.9 above. Rather surprisingly, a deterministic primality test (AKS) that runs in polynomial time has been found [**AKS04**]. In practice, however, the AKS test is not fast enough, so the probabilistic *Miller-Rabin test* is preferable. This test is based on the Proposition below. Note that in this section n represents any natural number, not a security parameter.

Proposition 9.10. *Let $n \in \mathbb{N}$ be odd and $1 \leq a < n$ an integer. If $\gcd(a,n) \neq 1$, then n is composite. Otherwise, write*

$$n - 1 = 2^s d$$

with $s \in \mathbb{N}$ being maximal. If n is prime, then either

$$a^d \equiv \pm 1 \mod n$$

or there is an integer $r \in \{1, \dots, s-1\}$ such that

$$a^{2^r d} \equiv -1 \mod n.$$

Proof. Suppose that n is a prime and $y = a^d \not\equiv 1 \mod n$. Consider the sequence

$$y,\ y^2,\ \dots,\ y^{2^{s-1}},\ y^{2^s} \mod n.$$

Euler's Theorem 4.15 gives

$$a^{n-1} = (a^d)^{2^s} = y^{2^s} \equiv 1 \mod n.$$

Hence the above sequence of squares starts with $y \not\equiv 1$ and ends with 1, but since we assumed that n is a prime so that $\mathbb{Z}_n$ is a field, the equation $x^2 \equiv 1 \mod n$ implies $x \equiv 1$ or $x \equiv -1$. Therefore, one of the elements in the above sequence must be congruent to -1. $\square$

The Miller-Rabin test checks whether n satisfies the implication of the above Proposition 9.10. If it does not, then n is composite. Hence all n satisfying the following condition (COMP) must be composite:

(COMP) *There exists a base $a \in \mathbb{N}$ with $1 \leq a < n$, such that*

$$a^d \not\equiv \pm 1 \mod n \text{ and } a^{2^r d} \not\equiv -1 \mod n \text{ for all } 1 \leq r \leq s-1.$$

This is used in Algorithm 9.1.

Algorithm 9.1 Miller-Rabin Algorithm with k runs

```
Input: $n \in \mathbb{N}, k \in \mathbb{N}$
Output: ‘$n$ is probably prime’ or ‘$n$ is composite’
for $i = 1$ to $k$ do
    $a \overset{\$}{\leftarrow} \{1, 2, \dots, n-1\}$ // choose a uniform random base $a$.
    if $\gcd(a, n) \neq 1$ then
        return ’$n$ is composite’
    end if
    $y \equiv a^d \mod n$
    if $y \equiv 1$ or $y \equiv -1$ then
        do next // $n$ could be a prime, test another base $a$.
    end if
    for $r = 1$ to $s - 1$ do
        $y = y^2 \mod n$
        if $y \equiv 1$ then
            return ‘$n$ is composite’ // $n$ cannot be a prime, stop.
        end if
        if $y \equiv -1$ then
            do next // $n$ could be a prime, test another base $a$.
        end if
    end for
    return ‘$n$ is composite’ // $n$ cannot be a prime, stop.
end for
return ‘$n$ is probably prime’
```

If the Miller-Rabin algorithm outputs that a number is *composite*, this result must be correct. However, there are bases a such that a composite number n is incorrectly identified as a prime in a run of the Miller-Rabin test.

Proposition 9.11. *Let $n \in \mathbb{N}$ be composite and $n - 1 = 2^s d$, where $s \in \mathbb{N}$ is maximal. Then the number of bases $a \in \{1, 2, \dots, n-1\}$ such that*

$$a^d \equiv \pm 1 \mod n \text{ or } a^{2^r d} \equiv -1 \mod n \text{ for } r \in \{1, \dots, s-1\}$$

is at most $\frac{n-1}{4}$. ◊

We refer to [**Sho09**] for a proof of this statement. The probability that one run of the Miller-Rabin test identifies a composite number as prime is therefore less than $\frac{1}{4}$. One can reduce the error probability to less than $\frac{1}{4^k}$ by k independent runs of the Miller-Rabin test. Note that Proposition 9.11 holds for all composite n. One can show that the error probability for randomly selected odd numbers n is much lower, and in practice less than 10 runs are sufficient. The test is also efficient for large numbers, and we note that a full factorization of $n-1$ is not required in order to find the maximal exponent s of the factor 2. The exponent s and hence the necessary number of exponentiations is usually small, and the running time is $O(\text{size}(n)^3)$.

Example 9.12. (1) $n = 561$. We choose $a = 2$ and have $\gcd(2, 561) = 1$. Then $n-1 = 560 = 2^4 \cdot 35$, so that $d = 35$ and $s = 4$. One computes $a^d = 2^{35} \equiv 263 \not\equiv \pm 1 \mod 561$, so the test continues. The next steps are $a^{2d} = 2^{70} \equiv 166$, $a^{4d} = 2^{140} \equiv 67$ and finally $a^{8d} = 2^{280} \equiv 1$. The sequence does not contain the residue class -1, and thus the Miller-Rabin test shows that 561 is composite. In fact, $561 = 3 \cdot 11 \cdot 17$.

(2) $n = 1009$. We choose $a = 3$, so that $\gcd(3, 1009) = 1$. We have $n - 1 = 1008 = 2^4 \cdot 63$, hence $d = 63$ and $s = 4$. We compute $a^d = 3^{63} \equiv 192 \not\equiv \pm 1 \mod 1009$, so the test continues. Then $a^{2d} \equiv 540$ and $a^{4d} \equiv 1008 \equiv -1 \mod 1009$. Hence $n = 1009$ could be a prime and another base a is chosen. Every run of the test will confirm the result, since 1009 is in fact a prime number.

9.5. Efficiency of RSA

In general, RSA encryption and decryption requires one exponentiation modulo N. Assuming that the size of the base, exponent and modulus is n, the time complexity is $O(n^3)$, if fast exponentiation or the square-and-multiple algorithm is used (see Section 3.3). Exponentiation is an efficient operation, but the running time is nonetheless significant. Such an operation can take some time on hardware with restricted resources, for example on smart cards, but a large number of modular exponentiations should also be avoided on standard computer systems.

In practice, the public exponent is often chosen to be $e = 2^{16} + 1$. This is relatively small, but still large enough to protect against low-exponent attacks. An encryption $m^e \mod N$ only requires 16 modular squarings and one additional multiplication. Hence 17 multiplications modulo N are sufficient, and the complexity is $O(n^2)$, which is much better than the general case. At the time of this writing, it is recommended to choose a modulus of at least 2048 bits.

The private exponent d has full length even if e is short. In fact, a short exponent d would be insecure (Wiener attack). However, the decryption $c^d \mod N$ can be accelerated by a factor of around 4 by using the *Chinese Remainder Theorem* 4.26 (CRT).

Recall that the CRT gives a decomposition $\mathbb{Z}_N \cong \mathbb{Z}_p \times \mathbb{Z}_q$ (Theorem 4.26). The idea is to perform separate decryptions in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ and map the resulting tuple back to $\mathbb{Z}_N$. Since the size of p and q is around half the size of N and the running time is cubic, an exponentiation modulo p or q is approximately $2^3 = 8$ times faster than an exponentiation modulo N. Because exponentiations modulo p *and* q are necessary, the speed-up is a factor of 4. The running time of the CRT computation is insignificant compared to the exponentiations.

Let c be a ciphertext. First, the ciphertext and the private exponent are reduced:

$$\begin{aligned} c_p &= c \mod p, & c_q &= c \mod q, \\ d_p &= d \mod (p-1), & d_q &= d \mod (q-1). \end{aligned}$$

Note that the exponent d is reduced modulo $p-1$ and $q-1$, respectively, and not modulo p and q (see Proposition 4.16). In fact, one has $\operatorname{ord}(\mathbb{Z}_p^*) = p-1$ and $\operatorname{ord}(\mathbb{Z}_q^*) = q-1$. In the next step, the decryption is done more efficiently modulo p and q:

$$m_p = c_p^{d_p} \mod p, \quad m_q = c_q^{d_q} \mod q;$$

m is finally computed using the Chinese Remainder Theorem. Consider the equation

$$1 = xp + yq.$$

$x, y \in \mathbb{Z}$ can be computed using the Extended Euclidean Algorithm on input p and q. It follows that $xp \equiv 1 \mod q$ and $yq \equiv 1 \mod p$. Now we obtain

$$m = m_q xp + m_p yq \mod N.$$

Example 9.13. Let $p = 29$, $q = 23$, $N = pq = 667$, $e = 3$, $d = 411$ be RSA parameters and $c = 416$ a ciphertext (see Example 9.5). We want to accelerate the decryption $m = c^d$ by using the Chinese Remainder Theorem. First, we reduce the ciphertext c and the exponent d:

$$\begin{aligned} c_p &= 416 \mod 29 \equiv 10, & c_q &= 416 \mod 23 \equiv 2, \\ d_p &= 411 \mod 28 \equiv 19, & d_q &= 411 \mod 22 \equiv 15. \end{aligned}$$

Then the plaintext is computed modulo p and modulo q. The exponents d_p and d_q are much smaller than $d = 411$:

$$m_p = 10^{19} \mod 29 \equiv 21, \; m_q = 2^{15} \mod 23 \equiv 16.$$

We use the Extended Euclidean Algorithm to find $x, y \in \mathbb{Z}$ such that $1 = x \cdot 29 + y \cdot 23$ (see Table 9.1).

Table 9.1. Extended Euclidean Algorithm on input 29 and 23.

$29 : 23 = 1$ rem. 6	$29 = 23 + 6$	$6 = 29 - 23$
$23 : 6 = 3$ rem. 5	$23 = 3 \cdot 6 + 5$	$5 = 23 - 3 \cdot 6$
$6 : 5 = 1$ rem. 1	$6 = 5 + 1$	$1 = 6 - 5$

We obtain $1 = (29-23) - (23 - 3 \cdot (29-23)) = 4 \cdot 29 - 5 \cdot 23$, so that $x = 4$ and $y = -5$. Finally, we compute the plaintext:

$$m = m_q xp + m_p yq = 16 \cdot 4 \cdot 29 + 21 \cdot (-5) \cdot 23 = -559 \equiv 108 \mod 667.$$

9.6. Padded RSA

We saw in Section 9.3 that plain RSA is not CPA-secure, even if the parameters p, q, e, d are appropriately chosen, because the scheme is *deterministic*. Furthermore, the ciphertext is *malleable* and chosen ciphertext attacks are possible.

An obvious approach is to add random data before encrypting. In the *Public Key Cryptography Standard* (*PKCS*) *#1 version 1.5* [**MKJR16**], a message m is transformed into a padded message M that has the byte length of the modulus N:

$$M = 00||02||r||00||m;$$

r is a random padding string of at least 8 nonzero bytes. The extra padding bytes reduce the maximum length of the plaintext m by 11 bytes. This is not critical since N is usually at least 128 bytes long. The padded message M is encrypted using plain RSA:

$$c = M^e \mod N.$$

For decryption, one computes $M = c^d \mod N$ and checks whether the first two bytes of M are equal to $00||02$. Otherwise, an error is returned. The plaintext m is then recovered by stripping off the padding data. Unfortunately, this scheme can be attacked with an adaptive chosen ciphertext attack. The adversary only needs to know whether or not specifically crafted ciphertexts have correct padding after decryption [**Bon99**]. This information can be exploited in order to decrypt a given ciphertext and, for example, to attack the handshake of the TLS protocol, if RSA encryption of the Pre-Master-Secret is used (*Bleichenbacher's attack*). The error messages or the response time of a TLS server can leak the information whether or not the padding is correct. To prevent this attack, TLS servers should not inform clients about padding errors and continue with the handshake.

Furthermore, the scheme is not CPA-secure if the random string r is short, for example 8 bytes long. In this case, an adversary could recover a plaintext if parts of it are known, for example leading zeros. If r is longer, say half the length of N, then it is conjectured that the scheme is CPA-secure, but since it is does not protect against chosen ciphertexts attacks (CCA), another more complex scheme is recommended: *Optimal Asymmetric Encryption Padding* (OAEP). We describe the OAEP variant standardized in PKCS #1 version 2.2 and in RFC 8017 [**MKJR16**] (see Figure 9.3).

The scheme requires two hash functions in order to generate pseudorandom output: a hash function H with output byte length h and a so-called *mask generating function MGF* with input length h and variable output length. RFC 8017 proposes a mask generating function called MGF1 based on a hash function H, for example SHA-2 (see Chapter 7). MGF1 is defined as

$$MGF1(seed,\ len) = H(seed||ctr)\ ||\ H(seed||ctr-1)\ ||\ \dots\ ||\ H(seed||0),$$

where $ctr = \lceil \frac{len}{h} \rceil - 1$. The integers $0, \dots, ctr$ are represented as binary strings of length four bytes. MGF1 only outputs the leading len octets.

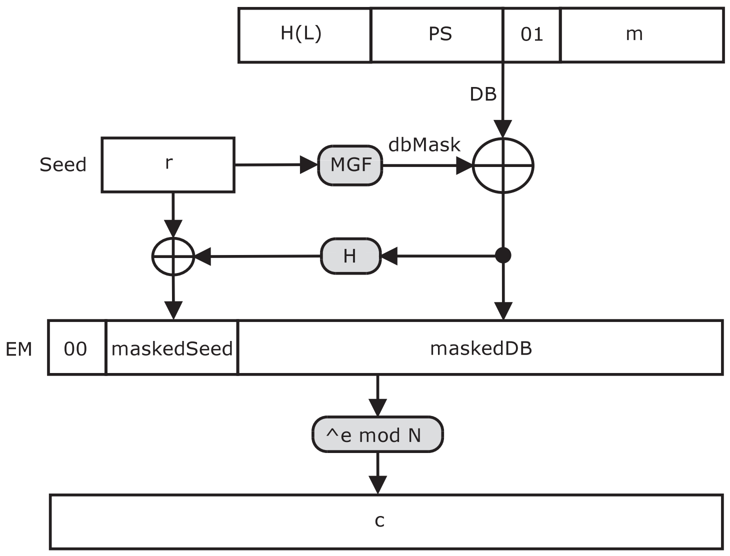


Figure 9.3. Encryption of a plaintext m using RSA-OAEP.

Let m be the plaintext message. The maximum byte length of m is $k-2h-2$, where k is the length of the modulus in bytes. Firstly, m is transformed into a data block DB of length $k-h-1$ bytes. One may add a label L or otherwise leave L empty. PS is a zero padding string of the required length. Then set

$$DB = H(L) \,||\, PS \,||\, \texttt{01} \,||\, m.$$

The data block DB can be viewed as a padded combination of the message and a hashed label.

Now the next step is to randomize the message. A random seed r of length h is generated, and $dbMask = MGF(r,\, k-h-1)$ gives a pseudorandom output string of length $k-h-1$ bytes. Define

$$maskedDB = DB \oplus dbMask,$$
$$maskedSeed = r \oplus H(maskedDB).$$

$maskedDB$ defines the randomized message, and $maskedSeed$ is needed during decryption to undo the masking of DB. The encoded message EM is given by

$$EM = \texttt{00} \,||\, maskedSeed \,||\, maskedDB.$$

The byte length of EM is k. Finally, the RSA-OAEP ciphertext is defined as

$$C = (EM)^e \mod N.$$

For decryption, $EM = C^d \mod N$ is computed and the length is checked. EM gives *maskedSeed* and *maskedDB*. Then compute

$$\begin{aligned} r &= maskedSeed \oplus H(maskedDB), \\ dbMask &= MGF(r,\, k-h-1), \\ DB &= maskedDB \oplus dbMask. \end{aligned}$$

The expected structure of DB and the label is verified, and finally the plaintext m is extracted. It is important that only one type of *decryption error message* is given for the different error conditions. Furthermore, the running time of OAEP implementations should not be correlated to the type of error. Otherwise, an adversary may obtain useful information and perform a chosen ciphertext attack.

Remark 9.14. A major result is that RSA-OAEP is *secure against adaptive chosen ciphertexts attacks* (CCA2-secure) under the RSA assumption and in the random oracle model [**FOPS01**]. However, the CCA2 security was proven for the original OAEP version of Bellare and Rogaway, not for the standardized version, which has a leading zero byte in the encoded message EM. Care must also be taken to ensure that an adversary cannot distinguish between the different error conditions. ◊

Loosely speaking, this result means that an adversary, who knows the public key and has access to a decryption oracle, cannot gain any information from a given ciphertext or tamper with a ciphertext.

9.7. Factoring

Factoring algorithms have been studied since the times of Ancient Greece, and many ideas have been contributed over the centuries, but no polynomial-time algorithm has been found and factoring is still assumed to be a hard problem, at least on conventional computers (see Chapter 13 on quantum computing). Obviously, if the factoring assumption turns out to be wrong, then RSA is broken.

In the following, we give an overview of different approaches to factoring and discuss their algorithmic complexity. We assume that a large positive integer $N = pq$ is given, where the prime factors p and q are unknown to an adversary.

Lists of primes up to a specified bound can be generated by the ancient *sieve of Eratosthenes*. The idea is to successively filter out all multiples of primes. There are faster modern algorithms, for example the *sieve of Atkin*. The sieve algorithm generates a list of prime numbers, but this is only efficient for relatively small primes.

Trial division is an elementary factoring method. It suffices to test numbers $\leq \sqrt{N}$. A list of small primes is useful (sieve method), and otherwise all odd numbers (or perhaps all numbers not divisible by 2, 3 or 5) need to be tested. The worst-case complexity is $O(\sqrt{N})$ and the running time is exponential in size(N).

Pollard's ρ algorithm searches for an integer x such that $\gcd(x, N)$ is either p or q, e.g., $x \equiv 0 \mod p$, but $x \not\equiv 0 \mod q$. The idea is to generate a pseudorandom sequence $x_i = f(x_{i-1})$ of integers modulo N and to find a collision modulo p or q using Floyd's cycle finding algorithm (see Proposition 1.63). Note that

$$x_k \equiv x_{2k} \mod p \Longleftrightarrow x_k - x_{2k} \equiv 0 \mod p.$$

The algorithm computes pairs x_k, x_{2k} of integers modulo N and checks whether $\gcd(x_k - x_{2k}, N) > 1$. By the birthday paradox, around $O(\sqrt{p}) = O(N^{1/4})$ iterations should be sufficient to find a collision modulo p.

Example 9.15. Let $N = 108371$. We choose the function $f(x) = x^2 + 1 \mod N$ and set $x_0 = 1$.

```
sage: N=108371
sage: def f(x):
          return(x*x+1)
sage: x=mod(1,N); y=mod(1,N)
      x=f(x)
      y=f(f(y))
      k=1
      while (gcd(x-y,N)==1):
          x=f(x)
          y=f(f(y))
          k=k+1
      print "k =",k,"x= ",x,"y= ",y,"p =",gcd(x-y,N)
k =  18 x=  21473 y=  67523 p = 307
```

We obtain the factor $p = 307$ after 18 iterations. The algorithm finds the collision $x_{18} = x_{36} \mod p$ and we verify that $21473 \equiv 67523 \equiv 290 \mod 307$. $\Diamond$

Fermat factorization uses a representation of N as a difference of squares:

$$N = x^2 - y^2 = (x + y)(x - y).$$

To find x and y, you begin with the integer $x = \lceil \sqrt{N} \rceil$ and increase x by 1 until $x^2 - N$ is square, say y^2, so that $N = x^2 - y^2$. Fermat factorization always works, since N can be written as a difference of two squares:

$$pq = \left(\frac{1}{2}(p+q)\right)^2 - \left(\frac{1}{2}(p-q)\right)^2 = x^2 - y^2.$$

However, Fermat's method is only efficient if the prime factors are close to one another, i.e., if y is small. In general, the running time is $O(\sqrt{N})$.

Example 9.16. Let $N = 14317$; then $\sqrt{N} \approx 119.7$. We begin with $x = 120$ and obtain $x^2 - N = 83$, which is not a square. Next, let $x = 121$ and now

$$x^2 - N = 324 = 18^2$$

is a square. Thus $y = 18$ and

$$N = (x + y)(x - y) = (121 - 18)(121 - 18) = 103 \cdot 139. \qquad \lozenge$$

The *quadratic sieve* generalizes Fermat factorization and is currently the fastest algorithm for numbers with less than around 100 decimal digits. One looks for integers x and y such that $x^2 \equiv y^2 \mod N$, but $x \not\equiv \pm y \mod N$. This implies

$$N \text{ divides } x^2 - y^2 = (x + y)(x - y),$$

but N divides neither $x+y$ nor $x-y$. Hence $\gcd(x-y,\ N)$ must be a non-trivial divisor of N and equal either p or q. The quadratic sieve tries to find suitable numbers x and y. It is reasonable to choose integers x close to $\sqrt{N}$, so that $x^2 - N$ is relatively small. Fermat factorization requires that x^2-N is a square number, but this is usually not the case. Now the idea is to multiply several (non-quadratic) numbers $x^2 - N$ with small prime factors (*smooth over a factor base*). The difficult task of the quadratic sieve is to find smooth numbers. Then one looks for a subset of smooth numbers such that their product is a square. A solution can be found using linear algebra over $GF(2)$, since a number is a square if the exponent of each prime factor is zero modulo 2.

The running time of the quadratic sieve is

$$O(e^{(1+o(1))\sqrt{\ln(N)\ \ln(\ln(N))}})$$

(see [**Pom96**]), where $o(1)$ is converging to 0 as $n \to \infty$. The algorithm is *sub-exponential*, but not polynomial.

Example 9.17. Let $N = 10441$; then $\sqrt{N} \approx 102.2$. We compute $x^2 - N$ for a couple of integers $x \geq 103$ and factorize the result. In general, complete factorization is inefficient and sieving should be used instead.

Suppose our factor base contains the primes $2, 3, 5, 7$. Then $x = 103, 104, 107, 109$ are interesting for this factor base, since x^2-N is smooth, i.e., divisible only by primes in the factor base.

```
sage: N=10441
sage: for x in range(103,110):
            s=factor(x^2-N)
            print x, x^2 - N, "=", s

103  168 = 2^3 * 3 * 7
104  375 = 3 * 5^3
105  584 = 2^3 * 73
106  795 = 3 * 5 * 53
107  1008 = 2^4 * 3^2 * 7
108  1223 = 1223
109  1440 = 2^5 * 3^2 * 5
```

Although $x^2 - N$ is not a square for any of these x and Fermat's method cannot be applied, their product is a square:

$$(103^2 - N) \cdot (104^2 - N) \cdot (107^2 - N) \cdot (109^2 - N) = 2^{12} \cdot 3^6 \cdot 5^4 \cdot 7^2.$$

Now we set $x = 103 \cdot 104 \cdot 107 \cdot 109$ and $y = 2^6 \cdot 3^3 \cdot 5^2 \cdot 7$ and obtain $x^2 \equiv y^2 \mod N$. We have $x \equiv 7491$, $y \equiv 10052$ modulo N and get $\gcd(7491 - 10052, 10441) = 197$, which is in fact a divisor of $N = 53 \cdot 197$. $\Diamond$

At the time of writing, the *number field sieve* is the most efficient algorithm for factoring large integers. With massive computing resources, numbers with more than 200 digits and for example the *RSA Challenge* with 768 bits could be factored using this method. Algebraic number fields are extension fields $\mathbb{Q}(\alpha)$ of $\mathbb{Q}$, where α is a root of a polynomial over $\mathbb{Q}$. The number field sieve uses the rings $\mathbb{Z}[\alpha]$ instead of the integers $\mathbb{Z}$, but this topic goes beyond the scope of this book.

The heuristic complexity of the number field sieve is

$$O(e^{(c+o(1))\ln(N)^{1/3}\ \ln(\ln(N))^{2/3}}),$$

where $c = \sqrt[3]{\frac{64}{9}} \approx 1.92$ (see [**Pom96**]).

Example 9.18. Suppose the sieve algorithm requires $f(N) = e^{c\ln(N)^{1/3}\ \ln(\ln(N))^{2/3}}$ steps. Then the effective key length (the bit strength) of RSA is $\log_2(f(N))$. For 1024-bit RSA, i.e., for $N \approx 2^{1024}$ and $c \approx 1.92$, one obtains 'only' $\log_2(f(N)) \approx 86.7$ bits. $\Diamond$

Pollard's $p - 1$ method can be applied for factoring $N = pq$, if $p - 1$ or $q - 1$ decompose into a product of small primes. In this case, we can guess multiples k of $p-1$. One defines k as a product of sufficiently large powers of small primes. If $(p-1) \mid k$, then Euler's Theorem 4.15 implies

$$a^k \equiv 1 \mod p$$

for all integers a with $\gcd(a, p) = 1$. One chooses a small integer $a > 1$ and computes $a^k \mod N$. Since k can be very large, fast exponentiation or the square-and-multiply algorithm should be used. Finally, $\gcd(a^k - 1, N)$ gives either p (method successful) or N (failure).

Example 9.19. Let $N = 1241143$. Set $a = 2$ and try

$$k = 2^3 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 360360.$$

Of course, other products are also possible. We compute

$$2^k \mod N = 2^{360360} \equiv 861526 \mod 1241143.$$

Finally, we have

$$\gcd(2^k - 1, N) = \gcd(861525,\ 1241143) = 547.$$

The method is successful and we find $N = 547 \cdot 2269$.

The prime $p = 547$ is vulnerable to Pollard's $p-1$ method since

$$p - 1 = 546 = 2 \cdot 3 \cdot 7 \cdot 13$$

is a product of small primes. ◊

The *Elliptic curve factorization method* (ECM) is another interesting factoring algorithm with sub-exponential running time. ECM is suitable for finding prime factors with up to about 80 decimal digits, but it is less efficient than the quadratic sieve or the number field sieve method for larger divisors. We outline ECM in Section 12.4.

Since no polynomial-time algorithm is known, the factoring assumption is currently well-founded, but in the future, *quantum computers* will probably be able to factorize large integers. Quantum computing and Shor's factoring algorithm are explored in Chapter 13.

The relative success of the known factoring algorithms show that standard key lengths of symmetric ciphers, i.e., 128 to 256 bits, are not sufficient for RSA (see Example 9.18). With large resources, a modulus with up to around 1000 bits can be factored. At the time of this writing, the use of 2048-bit integers N is recommended for long-term security against (non-quantum computing) attacks (see [**BSI18**]). The prime factors p and q should have around the same size (1024 bits) and their difference $p - q$ should be large.

Furthermore, the use of *strong primes* is sometimes recommended. A prime p is called strong if it is sufficiently large and satisfies additional conditions – in particular that $p - 1$ and $p + 1$ contain a large prime factor. This should provide protection against certain factoring methods, for example Pollard's $p - 1$ method. However, the size, randomness and independence of primes are more important and it is currently assumed that tests on strong primes do not significantly increase the security of RSA.

9.8. Summary

- Public-key cryptosystems use a public key for encryption and a private key for decryption. Indistinguishable encryptions under a chosen plaintext attack (CPA security) or under an adaptive chosen ciphertext attack (CCA2 security) are important requirements.
- The plain RSA cryptosystem uses the product of two large prime numbers and the security relies on the difficulty to factorize a given product.
- The probabilistic Miller-Rabin algorithm can efficiently test the primality of large integers.
- The plain RSA cryptosystem has weaknesses and the padded and randomized RSA-OAEP scheme should be used instead. OAEP can achieve CCA2 security under certain assumptions.
- Factoring algorithms with sub-exponential runtime exist, but no polynomial-time algorithms are known. RSA is considered to be secure against non-quantum computers, if the prime factors are randomly chosen and are more than 1000 bits long.

Exercises

1. Explain why a deterministic public-key encryption scheme is insecure if the number of possible plaintexts is small.
2. Suppose that p is a prime. Define a public-key scheme with the encryption function $\mathcal{E}_k(m) = m^e \mod p$ for a public key $k = (e, p)$ and a plaintext $m \in \mathbb{Z}_p^*$. Give the private key and the decryption function. Show that this scheme is insecure.
3. Consider a plain RSA cryptosystem with modulus $N = 437$ and public exponent $e = 5$.
   (a) Encrypt $m = 100$.
   (b) Factorize N and determine the private key d.
   (c) Decrypt the ciphertext and check that the result is $m = 100$.
4. Suppose that $m \in \mathbb{Z}_N$ is chosen uniformly at random, where $N = pq$ and $\text{size}(p) = \text{size}(q) = n$. Show that the probability of $m \notin \mathbb{Z}_N^*$ is negligible.
5. Consider Example 9.5. Which ciphertexts can be decrypted by a low-exponent attack? Now suppose that $\text{size}(N) = 2048$ and $e = 2^{16} + 1$. Which plaintexts are vulnerable to a low-exponent attack?
6. Bob's public RSA key is $(e = 35, N = 323)$. Apply the plain RSA encryption scheme:
   (a) Encrypt the plaintext $m = 66$ with Bob's public key. Use the fast exponentiation method.

(b) Mallory eavesdrops two ciphertexts $c_1 = 26$ and $c_2 = 213$, which were sent to Bob, but he does not know the plaintexts m_1 and m_2. How can Mallory compute the ciphertexts corresponding to the plaintexts $m_1 m_2 \bmod N$ and $m_1 m_2^{-1} \bmod N$ without carrying out an attack?

(c) Mallory chooses $s = 5$ and computes $y = s^e \bmod N \equiv 23$. He wants to find out the plaintext m corresponding to the ciphertext $c = 104$. He asks Bob to decrypt the 'innocent' ciphertext $c' = yc \bmod N \equiv 131$ and gets the plaintext $m' = 142$. Why is Mallory now able to determine m without computing the private exponent d? Determine the plaintext m.

(d) Now conduct an attack against this RSA key. Factorize N and compute d.

7. A plaintext m is encrypted with three different RSA moduli $N_1 = 901$, $N_2 = 2581$ and $N_3 = 4141$ using the public exponent $e = 3$. The ciphertexts are $c_1 = 98$, $c_2 = 974$, $c_3 = 2199$. Conduct Hastad's broadcast attack and determine the plaintext m.
Tip: Set $N = N_1 N_2 N_3$ and find $c \bmod N$ such that $c = c_i \bmod N_i$ for $i = 1, 2, 3$; then compute $m = \sqrt[3]{c}$.

8. Side-channel attacks against RSA use the power consumption of an implementation to derive the private key. Suppose a microprocessor uses the square-and-multiply algorithm to decrypt a ciphertext with a private key d. An attacker analyzes the power trace and concludes that the decryption uses the following sequence of modular squarings (SQ) and multiplications (MULT): SQ, SQ, SQ, SQ, SQ, MULT, SQ, MULT, SQ, SQ, SQ, SQ, MULT, SQ, MULT.

(a) Determine the private key d.
Tip: Use the construction of the square-and-multiply algorithm given in Chapter 3.

(b) The public key is $(e = 11, N = 8051)$. Calculate $\varphi(N)$, p and q from d, e and N and verify your result.

9. The *Fermat primality test* of $n \in \mathbb{N}$ chooses a uniform random integer $a \in \{1, \ldots, n-1\}$, computes $a^{n-1} \bmod n$ and outputs *n is composite*, if the result is not congruent to 1. Otherwise, the test outputs *n is probably prime*. Show that the test is correct. However, there are composite numbers n which are identified as possible primes for all $a \in \mathbb{Z}_n^*$. They are called *Carmichael* numbers. Show that $n = 561$ is a Carmichael number.

10. Check the primality of $n = 263$ using the Miller-Rabin algorithm and $a = 3$ as well as $a = 5$.

11. Encrypt $m = 2314$ with the plain RSA cipher and the public key $(e = 5, N = 10573)$. Factorize N using Fermat's method. Why is $e = 5$ an admissible exponent, whereas $e = 3$ is not permitted? Determine the corresponding private key d. Decrypt the ciphertext and check the result. Use the Chinese Remainder Theorem to reduce the size of the exponents.

12. Two RSA moduli are given: $N_1 = 101400931$ and $N_2 = 110107021$. They have a common prime factor. Show that both RSA keys are insecure and compute the factorization of N_1 and N_2.
13. An adversary is able to modify a RSA ciphertext. They want to square the unknown plaintext modulo N. Why is this attack possible for plain RSA, but not if RSA-OAEP is used?
14. Let $(e = 5, N = 10057)$ be the public key of an RSA cryptosystem. Encrypt the message $m = 2090$ using the plain RSA scheme. Factorize N and find the decryption exponent d.
15. Assume that RSA with a modulus of length 1024 bits and the encryption exponent $e = 2^{16} + 1$ is used. How many modular multiplications are needed, at most, for encryption and for decryption?
16. Factorize $N = 2041$ using the quadratic sieve method.
    Remark: This example is discussed in [**Pom96**].
17. Factorize $N = 10573$ with Pollard's $p - 1$ method. Choose $a = 2$ and try $k = 2^3 3^3$; then give reasons why this attack is successful for the given integer N.
18. Describe an attack against RSA encryption with random padding if the padding string is short and the number of possible plaintexts is small.

Chapter 10

Key Establishment

Keys play a crucial role in cryptography and the establishment of secret keys between two (or more) parties is a non-trivial task. A key establishment method should preferably not require a secure channel and provide protection against adversaries.

Key distribution by a trusted authority is briefly discussed in Section 10.1. Key exchange or key agreement is a method where the parties exchange messages and jointly generate a secret key. We explain key exchange protocols and discuss their security requirements in Section 10.2. A widely used method is the Diffie-Hellman key exchange, which is dealt with in Section 10.3. Diffie-Hellman is a public-key scheme, which uses a large cyclic group in which the discrete logarithm is hard to compute. The most important example is the multiplicative group $\mathbb{Z}_p^*$ of integers modulo a prime number, and this is explained in Section 10.4. Another possibility is the group of points on an elliptic curve over a finite field which is discussed in Section 12.2. In Section 10.5, we present algorithms to solve the discrete logarithm problem and discuss their complexity.

Key encapsulation forms an alternative to key exchange and is covered in Section 10.6. There is also an encapsulation variant of the Diffie-Hellman key exchange. The combination of key encapsulation and symmetric encryption gives hybrid public-key encryption schemes, which are outlined in Section 10.7.

Key establishment, the Diffie-Hellman key exchange and the discrete logarithm problem are standard topics in many cryptography textbooks, for example [**PP10**]. A discussion of various methods for key distribution and key management can be found in [**GB08**]. Refer to [**KL15**] for further details on security definitions and proofs of key exchange, key encapsulation and hybrid encryption schemes.

10.1. Key Distribution

The secure establishment of secret keys is a major prerequisite of symmetric cryptography. Until the 1970s, it was assumed that an initial confidential and authenticated channel was necessary for key distribution. The channel is used to establish a long-term secret key between two parties in a *key predistribution* scheme. After the initial setup, the long-term key can be used to encrypt a short-term session key that is sent over a channel that may be insecure.

The distribution of keys can be delegated to a *trusted third party* or a central instance. With appropriate protocols, it is sufficient to share a long-term secret key with a *Key Distribution Center* (KDC). Basically, the KDC generates and encrypts a random session key under the long-term secret key of the involved parties. Then the KDC sends the encrypted session key to the destination (see Figure 10.1). The parties decrypt the session key and leverage it to protect their communication.

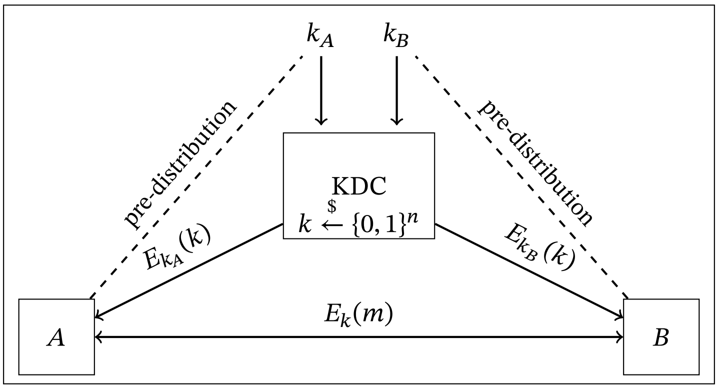


Figure 10.1. Key distribution using a trusted third party.

We note that the basic protocol described above is only secure against eavesdropping and not against active attacks. *Kerberos* is an example of a more advanced and widely used key distribution protocol (see [**GB08**] and RFC 4120 [**NYHR05**]).

The question of how to bootstrap the key distribution and establish long-term secret keys remains. In the following section, we define the security requirements of key exchange protocols that do not assume a secure channel ahead of time. In Section 10.3, we will see that the Diffie-Hellman key exchange is an important example of such a protocol.

10.2. Key Exchange Protocols

A key exchange (or key agreement) protocol is a distributed algorithm between two (or more) parties who exchange messages and finally compute a secret key. Now, we do not assume a pre-distribution of keys or a secure channel between the parties. Nevertheless, the protocol should be secure against eavesdropping attacks.

Definition 10.1. Suppose a key exchange protocol is given. Consider the following experiment (see Figure 10.2). Two communication parties (Alice and Bob) hold 1^n, exchange the messages m and derive a key k of length n. A challenger chooses a random bit $b \overset{\$}{\leftarrow} \{0,1\}$. If $b = 1$ set $k' = k$, and otherwise $k' \overset{\$}{\leftarrow} \{0,1\}^n$ is chosen uniformly at random. An adversary A is given 1^n, the transcript m and the challenge k'. They try to guess b, i.e., to distinguish between the secret key k and a random string, and output a bit b'. The challenger outputs 1 if $b = b'$, and 0 otherwise. The key exchange (KE) advantage of A is defined as

$$\mathrm{Adv}^{\mathrm{KE-eav}}(A) = |\, Pr[b' = b] - Pr[b' \neq b] \,| \,.$$

The key exchange protocol is *secure in the presence of an eavesdropper* (EAV-secure) if, for every probabilistic polynomial time adversary A, the advantage $\mathrm{Adv}^{\mathrm{KE-eav}}(A)$ is negligible in n. ◊

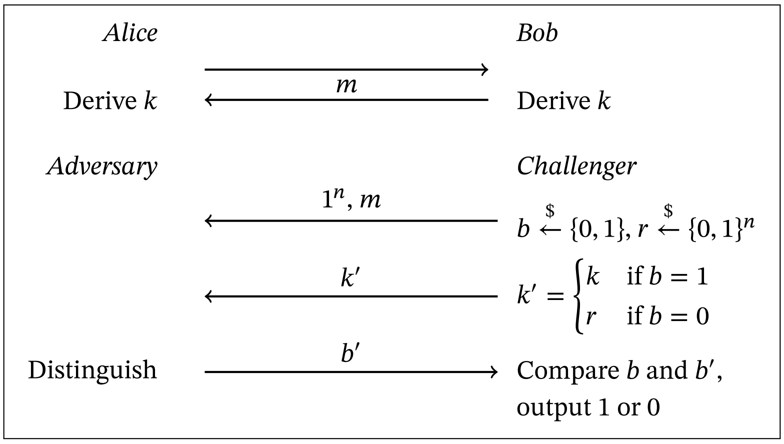

Alice
Bob
Derive k
m
Derive k
Adversary
Challenger
$1^n, m$
$b \overset{\$}{\leftarrow} \{0,1\}, r \overset{\$}{\leftarrow} \{0,1\}^n$
k'
$k' = \begin{cases} k & \text{if } b = 1 \\ r & \text{if } b = 0 \end{cases}$
Distinguish
b'
Compare b and b', output 1 or 0

Figure 10.2. Key distinguishability experiment.

The above definition of EAV security requires that the protocol messages m do not reveal a single bit of information on the key k to an eavesdropper. Otherwise, the adversary would be able to distinguish between k and a random string.

Remark 10.2. Note that the above experiment assumes a passive attacker who is unable to change or inject any messages. The presence of active adversaries requires an *authenticated key exchange* (AKE) protocol, where the communication partners are able to verify the authenticity of messages. Yet another topic is *perfect forward secrecy* (PFS), which guarantees the security of past session keys if long-lived keys are exposed.

10.3. Diffie-Hellman Key Exchange

The Diffie-Hellman protocol was a breakthrough in cryptography, because it solved the problem of secure key exchange over an insecure channel without pre-distribution of secret keys. The protocol was the first published public-key algorithm [**DH76**] and the starting point of a new type of cryptography.

In this section, we explain the protocol for an arbitrary cyclic group G. A standard choice of G are (subgroups of) the multiplicative group $\mathbb{Z}_p^*$ of integers modulo a prime number p (see Section 10.4). The multiplicative groups $GF(2^m)^*$ can also be used, but research has shown that they are less secure than groups $\mathbb{Z}_p^*$ of the same size. Another option is the group of points on an elliptic curve over a finite field. We explore the elliptic curve Diffie-Hellman key exchange in Section 12.2.

The Diffie-Hellman key exchange protocol requires a cyclic group G of order q and a generator $g \in G$. The parameters (G, q, g) are public and have to be exchanged between the communication parties Alice and Bob if they are not known in advance. Alice generates a private uniform random number $a \in \mathbb{Z}_q$, i.e., a positive integer less than q, and sends $A = g^a \in G$ to Bob. Bob also chooses a private uniform element $b \in \mathbb{Z}_q$ and sends $B = g^b \in G$ to Alice. The communication channel between Alice and Bob can be public. Alice derives the shared secret key by computing $k = B^a \in G$ and Bob computes $k = A^b \in G$ (compare Figure 10.3). The scheme is correct, since B^a and A^b are both equal to $g^{ab} \in G$.

$$\begin{array}{ccc} \textit{Alice} & \xrightarrow{\quad G, q, g \quad} & \textit{Bob} \\ a \overset{\$}{\leftarrow} \mathbb{Z}_q,\ A = g^a & \xrightarrow{\quad A \quad} & b \overset{\$}{\leftarrow} \mathbb{Z}_q,\ B = g^b \\ k = B^a & \xleftarrow{\quad B \quad} & k = A^b \end{array}$$

Figure 10.3. Diffie-Hellman key exchange between Alice and Bob.

Note that the result of the Diffie-Hellman key exchange is a *group element*, not a binary string. In practice, one applies a *key derivation function* to k, which transforms the group element into a binary string.

The security of the Diffie-Hellman key exchange is closely related to the *discrete logarithm* (DL) problem. If g is a generator of G and $\mathrm{ord}(G) = \mathrm{ord}(g) = q$, then

$$G = \{e,\ g^1,\ \ldots,\ g^{q-1}\}.$$

We therefore have a bijection between the elements of G and the exponents $0, 1, \ldots, q-1$. For each $h \in G$, we call the corresponding exponent the *discrete logarithm* of h to the base g and write $\log_g(h)$. One has

$$g^{\log_g(h)} = h.$$

In the Diffie-Hellman protocol, the exponents $a = \log_g(A)$ and $b = \log_g(B)$ are kept secret. One assumes that an eavesdropper is not able to efficiently compute discrete logarithms of A. Of course, some discrete logarithms, for example $\log_g(e) = 0$ and $\log_g(g) = 1$, are obvious and pre-computed tables of powers g^a can also be used. The discrete logarithm problem is called *hard* relative to the generated parameters if for any probabilistic polynomial-time adversary A, there is only a negligible probability (in terms of the group size) that an adversary finds the discrete logarithm $\log_g(A)$ of a *uniform* group element A.

The *Computational Diffie-Hellman* (CDH) problem is to compute the Diffie-Hellman shared secret $k = g^{ab} = A^b = B^a$ for given uniform elements $A, B \in G$. The CDH problem is not harder than the discrete logarithm (DL) problem. In other words, a solution to the DL problem also solves the CDH problems.

Now, the security of the Diffie-Hellman protocol relies on the *Decisional Diffie-Hellman* (DDH) problem defined below. The DDH problem is not harder than the CDH and the DL problem.

Definition 10.3. Consider the following experiment (see Figure 10.4): on input 1^n, a cyclic group $G = \langle g \rangle$ of order q is generated where size$(q) = n$. Choose random numbers $a \overset{\$}{\leftarrow} \mathbb{Z}_q$, $b \overset{\$}{\leftarrow} \mathbb{Z}_q$, compute $A = g^a$, $B = g^b$, $k = g^{ab}$ and choose a random bit $b \overset{\$}{\leftarrow} \{0,1\}$. An adversary obtains G, q, g, A, B and k (if $b = 1$) or a uniform random element $r \overset{\$}{\leftarrow} G$ (if $b = 0$). The adversary tries to guess b and outputs a bit b'. The output of the experiment is 1 if $b' = b$, and 0 otherwise.

The DDH-advantage of A is

$$\mathrm{Adv}^{\mathrm{DDH}}(A) = |Pr[b' = b] - Pr[b' \neq b]\,|.$$

The DDH problem is hard relative to the generation of group parameters, if for every probabilistic polynomial time adversary A, the advantage $\mathrm{Adv}^{\mathrm{DDH}}(A)$ is negligible in terms of n. $\diamondsuit$

If the DL problem or the CDH problem is easy, then the DDH problem is easy, too, but the converse is not known. The DDH assumption is therefore stronger than the DL assumption.

Theorem 10.4. *If the DDH problem is hard relative to the generation of group parameters, then the Diffie-Hellman key exchange protocol is secure in the presence of an eavesdropper (EAV-secure).*

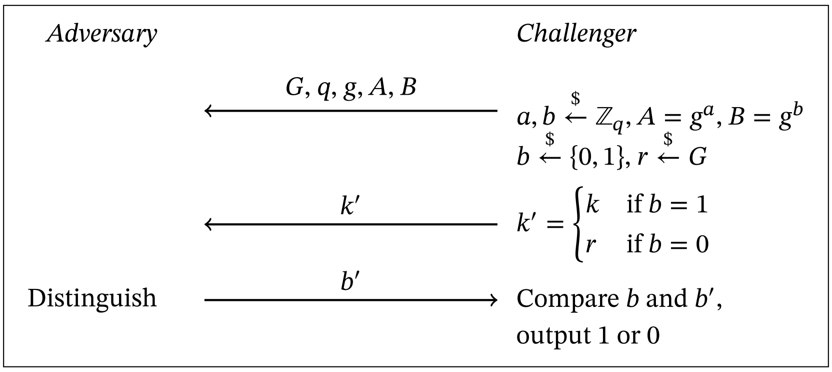

Adversary
Challenger
G, q, g, A, B
$a, b \overset{\$}{\leftarrow} \mathbb{Z}_q, A = g^a, B = g^b$
$b \overset{\$}{\leftarrow} \{0,1\}, r \overset{\$}{\leftarrow} G$
k'
$k' = \begin{cases} k & \text{if } b = 1 \\ r & \text{if } b = 0 \end{cases}$
Distinguish
b'
Compare b and b', output 1 or 0

Figure 10.4. Decisional Diffie-Hellman (DDH) experiment.

Remark 10.5. It is not difficult to show that solving the above DDH experiment and distinguishing a Diffie-Hellman shared secret k from a uniform random element are equivalent problems (see [**KL15**]). Note that k is an element of group G. The above Theorem therefore requires a slightly modified key distinguishability experiment: the key and the random element are from group G instead of an n-bit string. Alternatively, one applies a key derivation function to transform the shared secret key into a binary string.

Remark 10.6. It is important to observe that the plain Diffie-Hellman protocol does not protect against *active* adversaries. If an attacker is able to replace A and B with their own parameters, then they can perform a *Man-in-the-Middle attack*. The problem of the plain Diffie-Hellman protocol is the lack of *authenticity* (see Remark 10.2). In practice, one often signs the public keys in order to prove their authenticity. Signatures are covered in Chapter 11. However, some issues remain, because at some point a trusted public key (a trust anchor) is needed.

10.4. Diffie-Hellman using Subgroups of $\mathbb{Z}_p^*$

The multiplicative group $\mathbb{Z}_p^*$ and its subgroups are often used in a Diffie-Hellman key exchange. In fact, the original construction in [**DH76**] was based on this type of group. The idea is that the modular power function $f(x) = g^x \bmod p$ is a *one-way permutation*, which is easy to compute but is hard to invert if the parameters p and g are properly chosen.

If p is a prime, then $\mathbb{Z}_p^* = GF(p)^*$ is a cyclic group of order $p - 1$ (see Theorem 4.24). Any $g \in \mathbb{Z}_p^*$ generates a cyclic subgroup G of order $q = \text{ord}(g) \mid p - 1$ and could be used in the protocol, but since the discrete logarithm problem must be hard, q should be large. However, this condition is not sufficient if q can be factorized into a product of small primes. For the hardness of the discrete logarithm and the DDH problem, it is advisable for q to be a *large prime*.

Suppose that h is a generator of $\mathbb{Z}_p^*$, i.e., $\operatorname{ord}(h) = p-1$, and $p-1 = rq$, where q is a large prime. Then $\operatorname{ord}(h^r) = \frac{p-1}{r} = q$ and $G = \langle h^r \rangle$ is a cyclic group of prime order q. We let $g = h^r$ and thus obtain Diffie-Hellman parameters.

Furthermore, *safe primes* are useful in the generation of Diffie-Hellman parameters. A prime p is called safe if $q = \frac{p-1}{2}$ is prime. Then $p-1 = 2q$ and the order of any element $g \in \mathbb{Z}_p^*$ with $g \not\equiv \pm 1$ is either $p-1$ or q.

Example 10.7. Let $p = 59$. Of course, p is much too small to be secure. One has $\operatorname{ord}(\mathbb{Z}_p^*) = p-1 = 2 \cdot 29$, so 59 is a safe prime. We are looking for a generator of $\mathbb{Z}_p^*$ and try $h = 2 \mod 59$. In fact, $\operatorname{ord}(h) = 58$, i.e., h is a primitive root mod 59, since

$$h^2 = 4 \not\equiv 1 \text{ and } h^{29} \equiv 58 \not\equiv 1 \mod 59$$

(see Algorithm 4.1). Hence h generates the full multiplicative group of order 58 and $g = h^2$ generates a subgroup of prime order 29.

(1) We perform a Diffie-Hellman key exchange with the parameters $G = \mathbb{Z}_{59}^*$, $h = 2 \mod 59$ and $\operatorname{ord}(h) = 58$. Alice selects $a = 7$ and sends $A = 2^7 \mod 59 \equiv 10$ to Bob. Bob chooses $b = 24$ and transmits $B = 2^{24} \mod 59 \equiv 35$ to Alice. Alice computes $k = B^a = 35^7 \mod 59 \equiv 12$ and Bob obtains the same key $k = A^b = 10^{24} \mod 59 \equiv 12$.

(2) Set $g = h^2 = 4 \mod 59$ and $G = \langle 4 \rangle \subset \mathbb{Z}_{59}^*$ and perform a Diffie-Hellman exchange using G, g and $q = \operatorname{ord}(g) = 29$. Alice selects $a = 7$ and sends $A = 4^7 \mod 59 \equiv 41$ to Bob. Bob chooses $b = 24$ and sends $4^{24} \mod 59 \equiv 45$ to Alice. Alice computes $k = 45^7 \mod 59 \equiv 26$ and Bob gets $k = 41^{24} \mod 59 \equiv 26$. ◊

In practice, standardized parameters G, g and q are used. A set of pre-defined parameters is called a *Diffie-Hellman group*. Note that – in contrast to RSA – the Diffie-Hellman parameters G, g and q *can be re-used*. However, every key exchange should use fresh uniform random exponents a and b (see Exercise 10).

Example 10.8. RFC 7919 [**Gil16**] defines a 2048-bit Diffie-Hellman group:

```
p = FFFFFFFF FFFFFFFF ADF85458 A2BB4A9A AFDC5620 273D3CF1
    D8B9C583 CE2D3695 A9E13641 146433FB CC939DCE 249B3EF9
    7D2FE363 630C75D8 F681B202 AEC4617A D3DF1ED5 D5FD6561
    2433F51F 5F066ED0 85636555 3DED1AF3 B557135E 7F57C935
    984F0C70 E0E68B77 E2A689DA F3EFE872 1DF158A1 36ADE735
    30ACCA4F 483A797A BC0AB182 B324FB61 D108A94B B2C8E3FB
    B96ADAB7 60D7F468 1D4F42A3 DE394DF4 AE56EDE7 6372BB19
    0B07A7C8 EE0A6D70 9E02FCE1 CDF7E2EC C03404CD 28342F61
    9172FE9C E98583FF 8E4F1232 EEF28183 C3FE3B1B 4C6FAD73
    3BB5FCBC 2EC22005 C58EF183 7D1683B2 C6F34A26 C1B2EFFA
    886B4238 61285C97 FFFFFFFF FFFFFFFF
g = 2
```

The group order is $q = \frac{p-1}{2}$. ◊

Finally, we briefly discuss the *efficiency* of Diffie-Hellman. The Diffie-Hellman key exchange requires two *exponentiations* by each communication partner. The public keys $A = g^a$ and $B = g^b$ can be pre-computed if the group and the generator are known in advance. Then the shared secret k can be determined with only one exponentiation by each party. The modulus is p, but the exponent is bounded by q. Therefore, the running time of the modular exponentiations is $O(\text{size}(p)^2 \text{ size}(q))$.

10.5. Discrete Logarithm

We have seen that solving the discrete-logarithm (DL) problem breaks the Diffie-Hellman key exchange. In the following, we give a brief overview of existing algorithms to compute discrete logarithms.

Suppose that a cyclic group G (not necessarily a subgroup of $\mathbb{Z}_p^*$), a generator g of order q, $n = \text{size}(q)$ and $A = g^a$ are given. We are looking for algorithms computing the exponent a. Note that we use the *multiplicative* notation for the group G.

Obviously, one can compute $g^x \in G$ for all exponents $0 \leq x < q$ and compare the result with A. The complexity of this approach is $O(q) = O(2^n)$ (in the worst case) and hence exponential. This can be reduced to $O(\sqrt{q}) = O(2^{n/2})$, which is still exponential, using *Shank's Babystep-Giantstep* algorithm: let $m = \lfloor \sqrt{q} \rfloor$; then the unknown exponent a can be written as

$$a = ms + r, \text{ where } r < m.$$

Since $A = g^a$, one has $A = g^{ms} g^r$ or, equivalently, $Ag^{-r} = (g^m)^s$. The elements Ag^{-r} for $0 \leq r < m$ are called *babysteps* and have to be stored. If one of the babysteps equals 1, then set $a = r$ and the problem is solved. Otherwise, set $T = g^m$, compute the *giantsteps* T^s for $0 < s \leq m$ and compare them to the babysteps. If the giantstep T^s is equal to the babystep Ag^{-r}, then the solution to the DL problem is $a = ms + r$. In the worst case, all babysteps and giantsteps have to be computed, which requires $2m$ exponentiations. Furthermore, m babysteps have to be stored. Hence the running time and the space complexity is $O(2^{n/2})$.

Example 10.9. Let $p = 59$ and $g = 4 \in \mathbb{Z}_p^*$; then $q = \text{ord}(g) = 29$. Suppose an adversary eavesdrops $A = 41$ (see Example 10.7). They compute the discrete logarithm using the Babystep-Giantstep algorithm: $m = \lfloor \sqrt{29} \rfloor = 5$, $g^{-1} = (4 \bmod 59)^{-1} \equiv 15$. The babysteps are:

$$A = 41,\ Ag^{-1} = 25,\ Ag^{-2} = 21,\ Ag^{-3} = 20,\ Ag^{-4} = 5.$$

Furthermore, $T = g^m = 4^5 \bmod 59 \equiv 21$. Hence the first giantstep matches the second babystep and the solution is $a = 1 \cdot 5 + 2 = 7$. In fact, $g^a = 4^7 \bmod 59 \equiv 41$. ◊

Pollard's ρ method for logarithms has about the same running time $O(\sqrt{q}) = O(2^{n/2})$ as the Babystep-Giantstep algorithm, but requires much less storage. We briefly outline the approach: Generate a sequence of elements $g^x A^y \in G$, where $(x, y) \in \mathbb{Z}_q \times \mathbb{Z}_q$, and use Floyd's cycle finding algorithm (see Proposition 1.63) to find a collision

$$g^{x_1} A^{y_1} = g^{x_2} A^{y_2}.$$

Since G is cyclic of order q, the exponents satisfy $x_1 + ay_1 \equiv x_2 + ay_2 \mod q$, so that a is equal to

$$a = \frac{x_2 - x_1}{y_1 - y_2} \mod q.$$

The *Pohlig-Hellman algorithm* can be applied if $q = \operatorname{ord}(G)$ is a product of powers of *small primes*. By Theorem 4.29, every finite abelian group is isomorphic to a product of cyclic groups of prime-power order. The discrete logarithm can be computed separately for each of the factors, and the Chinese Remainder Theorem combines this into a result for the full group. Furthermore, one can speed up the computation of discrete logarithms in groups of prime-power order. We refer to [**Sho09**] for more details. Therefore, q should be a prime or at least contain a large prime divisor.

Example 10.10. Let $p = 73$ and $g = 11 \mod 73$. One can easily show that g is a generator of $G = \mathbb{Z}_p^*$. Suppose $A = 62 \mod 73$ is given and we want to compute $\log_g(A)$. Since $\operatorname{ord}(G) = 72 = 8 \cdot 9$, one has an isomorphism

$$G \cong G_1 \times G_2,$$

where G_1 and G_2 are cyclic subgroups of order 8 and 9, respectively. The isomorphism maps $h \in G$ to $(h^9, h^8) \in G_1 \times G_2$. The image of g in $G_1 \times G_2$ is given by

$$g^9 = 11^9 \mod 73 \equiv 22\,,\ g^8 = 11^8 \mod 73 \equiv 2,$$

and the image of A in $G_1 \times G_2$ is given by

$$A^9 = 62^9 \mod 73 \equiv 51 \text{ and } A^8 = 62^8 \mod 73 \equiv 2.$$

Now we can compute the discrete logarithm in the smaller subgroups $G_1 = \langle 22 \rangle$ and $G_2 = \langle 2 \rangle$:

$$\log_{22}(51) = 5 \text{ and } \log_2(2) = 1.$$

The discrete logarithm in the full group G is a positive integer a which satisfies

$$a \equiv 5 \mod 8 \text{ and } a \equiv 1 \mod 9.$$

By the Chinese Remainder Theorem 4.26, these congruences have a unique solution modulo 72. The Extended Euclidean Algorithm on input 8 and 9 yields the equation $1 = -8 + 9$, and thus

$$a = 5 \cdot 9 + 1 \cdot (-8) = 37.$$

This is indeed the discrete logarithm of $A = 62$, since $g^a \equiv 11^{37} \mod 73 \equiv 62$. ◊

The *Index-Calculus algorithm* fixes a factor base B consisting of small primes and first computes $\log_g(p_i)$ for all $p_i \in B$. Then random integers $0 < x < p$ are chosen until $g^x A \mod p$ is a product of primes in the factor base. If this is the case, then $\log_g(g^x A)$

can be determined using the pre-computed discrete logarithms $\log_g(p_i)$. Finally, one obtains

$$\log_g(A) = \log_g(g^x A) - x \mod p - 1.$$

The algorithm can only be applied to the multiplicative groups of finite fields (and to some families of elliptic curves). The expected running time is

$$O(e^{(\sqrt{2}+o(1))\sqrt{\ln(p)\ln(\ln(p))}}).$$

The *Number Field Sieve for Discrete Logarithms* is currently the best available algorithm for the multiplicative group, and its heuristic sub-exponential running time is $O(e^{(c+o(1))\ln(p)^{\frac{1}{3}}\ln(\ln(p))^{\frac{2}{3}}})$, where $c = (\frac{64}{9})^{1/3} \approx 1.92$ (see [**JOP14**]). Note that the running time depends on the size of p and not on the size of the group order q. The effective key length for a 1024-bit prime p is only around 86 bits, and at the time of this writing primes of at least 2000 bits are recommended.

Polynomial-time algorithms are not known, but the discrete logarithm problem can be efficiently solved with *quantum computers* (see Chapter 13).

10.6. Key Encapsulation

Key encapsulation is a mechanism where a *public-key scheme* is leveraged to establish a secret key over an insecure channel. The sender encapsulates a randomly chosen secret key using the public key of the receiver and the receiver decapsulates the symmetric key using their private asymmetric key (see Figure 10.5).

Example 10.11. Suppose a public-key encryption scheme is given. Bob possesses a key pair (pk, sk) and Alice has a copy of his public key pk. Now Alice generates a uniform random symmetric key k and encrypts k using pk. The ciphertext $c = \mathcal{E}_{pk}(k)$ is sent to Bob, who decrypts c and recovers k using his private key sk. ♢

This is an example of a *Key Encapsulation Mechanism* (KEM), but the definition given below will be more general. Encapsulation and decapsulation is not necessarily identical to encryption and decryption, respectively. Furthermore, padding and randomization in encryption schemes such as RSA-OAEP may not be required for key encapsulation purposes.

Definition 10.12. A *key encapsulation mechanism* (KEM) consists of the following probabilistic polynomial-time algorithms (see Figure 10.5):

- A key generation algorithm $Gen(1^n)$ takes a security parameter 1^n as input and outputs a key pair (pk, sk).
- The encapsulation algorithm $Encaps$ takes a public key pk and a security parameter 1^n as input. It outputs a ciphertext c and a key $k \in \{0,1\}^n$ or, more generally,

a key of length $l(n)$. We write $(c, k) \overset{\$}{\leftarrow} Encaps_{pk}(1^n)$, where c is public and k is secret.

- The decapsulation algorithm *Decaps* takes a private key *sk* and a ciphertext c as input. It outputs a key $k = Decaps_{sk}(c)$ or an error symbol $\perp$.

The KEM provides correct encapsulation, if for $(c, k) \overset{\$}{\leftarrow} Encaps_{pk}(1^n)$, one has $Decaps_{sk}(c) = k$. ◊

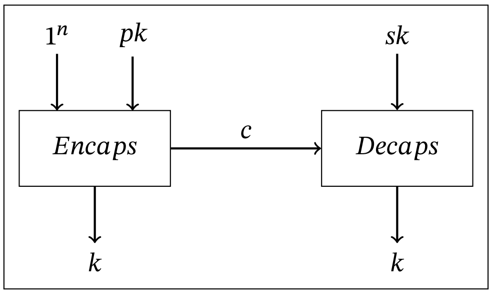


Figure 10.5. Key encapsulation and decapsulation.

A key encapsulation mechanism naturally defines a key establishment method: Alice takes Bob's public key *pk*, runs the encapsulation algorithm and keeps k. She sends the ciphertext c to Bob, who obtains k by running the decapsulation algorithm.

A KEM is secure under chosen plaintexts attacks (CPA-secure) if an adversary, who has access to *pk* and c, cannot distinguish between the encapsulated key k and a random string of the same length. CPA security means that an adversary does not learn a single bit of k from the ciphertext c.

Definition 10.13. Suppose a key encapsulation mechanism is given. Consider the following experiment: on input 1^n, the algorithm $Gen(1^n)$ generates a key pair (pk, sk). Then $Encaps_{pk}(1^n)$ is run and outputs (c, k). A uniform random bit $b \overset{\$}{\leftarrow} \{0, 1\}$ and a uniform random string $r \overset{\$}{\leftarrow} \{0, 1\}^n$ are chosen. If $b = 1$ then set $k' = k$, and otherwise $k' = r$. An adversary A is given (pk, c, k'), but the private key *sk* and b are not known to the adversary. A has to distinguish between the cases $k' = k$ and $k' = r$. The adversary outputs $b' = 1$ if they suspect that they obtained the key k and otherwise output $b' = 0$. The output of the experiment is 1 if $b' = b$, and 0 otherwise.

The IND-CPA advantage of the adversary A is defined as

$$\mathrm{Adv}^{\mathrm{ind-cpa}}(A) = |\, Pr[b' = b] - Pr[b' \neq b]\, | \,.$$

The scheme is CPA-secure if, for all probabilistic polynomial time adversaries A, the advantage $\mathrm{Adv}^{\mathrm{ind-cpa}}(A)$ is negligible in n. ◊

A stronger notion is security against *adaptive chosen ciphertext attacks* (CCA2 security). The corresponding experiment gives the adversary additional access to a *decapsulation oracle* (before and after obtaining the challenge), but they may not request the decapsulation of the challenge ciphertext c.

We construct a KEM based on RSA encryption, which can be shown to be CCA2-secure.

Definition 10.14. The *RSA key encapsulation mechanism* is defined as follows:

- The key generation algorithm $Gen(1^n)$ is identical to the RSA key generation (see Definition 9.4) and outputs a public key $pk = (e, N)$ and a private key $sk = (d, N)$. Furthermore, a hash function $H : \mathbb{Z}_N^* \to \{0,1\}^n$ is fixed.
- The encapsulation algorithm $Encaps$ takes the public key pk as input and chooses a uniform random element $s \in \mathbb{Z}_N^*$. It outputs the ciphertext
$$c = s^e \mod N$$
and the key $k = H(s)$.
- The decapsulation algorithm $Decaps$ takes c and the private key sk as input. It computes
$$s = c^d \mod N$$
and outputs $k = H(s)$. ◊

We infer from the RSA construction (see Definition 9.4) that the above encapsulation mechanism is correct. If the RSA assumption holds and the hash function behaves like a random oracle, then CPA security follows from the fact that an adversary is unable to derive s from c. But if s is unknown then $H(s)$ is uniform random. Note that padding schemes like OAEP are not required here since *s is uniform* in $\mathbb{Z}_N^*$. Furthermore, the RSA key encapsulation mechanism even turns out to be CCA2-secure if the hash function has no weaknesses. An adversary with access to a decapsulation oracle only gets the *hash value* $k' = H((c')^d \mod N)$ on input c'. However, this does not reveal any information about $k = H(c^d \mod N)$ if $c \neq c'$, since hashes of different input values are uncorrelated. We refer to [**KL15**] for a proof of the following Theorem:

Theorem 10.15. *If the RSA assumption holds and H is modeled as a random oracle, then the RSA key encapsulation mechanism is CCA2-secure.* ◊

The Diffie-Hellman key exchange protocol can also be turned into a key encapsulation mechanism. The Diffie-Hellman KEM can be viewed as an adaption of the ElGamal encryption scheme (see Exercise 12).

Definition 10.16. The *Diffie-Hellman key encapsulation mechanism* is defined by the following algorithms:

- The key generation algorithm Gen takes 1^n as input and outputs a cyclic group G of order q with $n = \text{size}(q)$, a generator $g \in G$, a uniform random element $b \in \mathbb{Z}_q$ and $B = g^b$. The public key is $pk = (G, q, g, B)$ and the private key is $sk = (G, q, g, b)$. Also fix a function $H : G \to \{0,1\}^n$.
- The encapsulation algorithm takes pk as input, chooses a uniform random element $a \in \mathbb{Z}_q$ and outputs the ciphertext $c = A = g^a$ and the key $k = H(B^a)$.
- The decapsulation algorithm $Decaps$ takes sk and c as input and outputs the key $k = H(c^b)$. ♢

The encapsulated key is $H(k')$, where k' is the shared Diffie-Hellman secret g^{ab}. Since $k' = g^{ab} = A^b = B^a$, the encapsulation method is correct. The security depends on standard assumptions about the Diffie-Hellman problem and on properties of H.

Theorem 10.17. *Suppose the computational Diffie-Hellman (CDH) problem is hard relative to the generation of group parameters and H is modeled as a random oracle. Then the Diffie-Hellman key encapsulation mechanism is CPA-secure.* ♢

The proof can be found in [**KL15**]. There is also a security guarantee without the use of a random oracle. Under the stronger *gap-CDH assumption* the Diffie-Hellman key encapsulation mechanism can be shown to be CCA2-secure. The gap-CDH problem (see [**OP01**]) gives the adversary access to a *Decisional Diffie-Hellman oracle* that answers whether (g, A, B, k') is a valid Diffie-Hellman quadruple, i.e., whether or not $k' = A^{\log_g(B)}$.

10.7. Hybrid Encryption

You may have noticed that we have not discussed public-key encryption of messages of *arbitrary length*. Multiple encryptions are of course possible, but are rarely used in practice because public-key schemes are not fast enough to process mass data. The standard approach is to combine public-key and symmetric-key encryption. The public-key scheme is only used for *key establishment* and the data is encrypted with a *symmetric algorithm*. This method is called *hybrid encryption*.

Definition 10.18. Suppose a key encapsulation mechanism (KEM) and a symmetric-key encryption scheme for messages of arbitrary length are given. Then a *hybrid encryption scheme* can be defined as follows:

- Run the key generation algorithm of the KEM on input 1^n and output the keys pk and sk.

- The hybrid encryption algorithm takes the public key pk and a message $m \in \{0,1\}^*$ as input. The encapsulation algorithm $Encaps$ computes

$$(c,k) \overset{\$}{\leftarrow} Encaps_{pk}(1^n).$$

  Then the symmetric encryption algorithm $\mathcal{E}$ takes k and the plaintext m as input and computes $c' = \mathcal{E}_k(m)$. Finally, output the ciphertext (c,c').
- The hybrid decryption algorithm takes the private key sk and the ciphertext (c,c') as input. First, the symmetric key is retrieved by computing

$$k = Decaps_{sk}(c).$$

  Then decrypt c' and output the plaintext $m = \mathcal{D}_k(c')$. If c or c' are invalid then output $\perp$. ♢

The advantage of hybrid schemes is that the computationally expensive public-key algorithm is only run once during encryption and decryption, independent of the length of the plaintext and the ciphertext.

We also want to give a security guarantee for a hybrid encryption scheme (see [**KL15**]).

Theorem 10.19. *Consider a hybrid encryption scheme as defined above.*

(1) *If the KEM is CPA-secure and the symmetric scheme is EAV-secure, then the corresponding hybrid scheme is CPA-secure.*

(2) *If the KEM and the symmetric scheme are both CCA2-secure, then the corresponding hybrid scheme is CCA2-secure.* ♢

Note that EAV security of the symmetric scheme is sufficient for (1). In fact, a hybrid scheme is public-key, and so EAV and CPA security are equivalent.

Corollary 10.20. *The hybrid encryption scheme that combines RSA key encapsulation and an authenticated encryption scheme (see Definition* 8.19) *is CCA2-secure if the RSA assumption holds and the hash function is modeled as a random oracle.* ♢

The Diffie-Hellman key encapsulation mechanism (see Definition 10.16) can be combined with a symmetric encryption scheme and a message authentication code. This defines the *Diffie-Hellman Integrated Encryption Scheme* (DHIES).

Definition 10.21. Suppose a symmetric encryption scheme and a message authentication code are given. Then the Diffie-Hellman Integrated Encryption Scheme (DHIES) is defined as follows.

- On input 1^n the key generation algorithm outputs a cyclic group G of order q with $n = \text{size}(q)$, a generator $g \in G$, a uniform random element $b \in \mathbb{Z}_q$ and $B = g^b$. The public key is $pk = (G, q, g, B)$ and the private key is $sk = (G, q, g, b)$. Furthermore, a function $H : G \to \{0,1\}^{2n}$ is fixed.
- The encryption algorithm takes a plaintext m and the public key pk as input, chooses a uniform random $a \in \mathbb{Z}_q$ and sets $k_E || k_M = H(B^a)$. Then compute $A = g^a$, $c \overset{\$}{\leftarrow} \mathcal{E}_{k_E}(m)$, $t = \text{MAC}_{k_M}(c)$ and output the ciphertext (A, c, t).
- The decryption algorithm takes the ciphertext (A, c, t) and the private key sk as input. Compute $k_E || k_M = H(A^b)$, verify the tag t using k_M and output the plaintext $m = \mathcal{D}_{k_E}(c)$. If $A \notin G$ or if the verification of t fails then output $\perp$. $\Diamond$

Note that the scheme derives both the symmetric encryption key and the message authentication key from the shared Diffie-Hellman secret.

DHIES usually refers to Diffie-Hellman using subgroups of $\mathbb{Z}_p^*$, the multiplicative group of integers modulo a prime number (see Section 10.4). However, if a group of points on an *elliptic curve* is used (see Section 12.2), then the scheme is called the *Elliptic Curve Integrated Encryption Scheme* (ECIES).

Diffie-Hellman integrated encryption schemes are CCA2-secure under certain assumptions:

Theorem 10.22. *Consider DHIES or ECIES and suppose the underlying symmetric encryption scheme is CPA-secure, the message authentication code is strongly secure, the gap-CDH assumption holds for the Diffie-Hellman group and the hash function H is modeled as a random oracle. Then DHIES and ECIES are CCA2-secure encryption schemes.* $\Diamond$

The above Theorem follows from the CCA2 security of the Diffie-Hellman key encapsulation method and the CCA2 security of the encrypt-then-authenticate construction (see Section 8.4) for CPA-secure symmetric encryption schemes.

Remark 10.23. Several of the above statements, in particular on the CCA2 security of key encapsulation and hybrid encryption schemes, require the assumption that the hash function is modeled as a random oracle. We refer to the literature for security guarantees without the use of the random oracle model ([**KL15**], [**HK07**]).

10.8. Summary

- The distribution of secret keys can be delegated to a Key Distribution Center. This requires the pre-distribution of long-term keys. Kerberos is a widely used example of a key distribution protocol.
- In a key exchange protocol, the communication partners exchange messages over an insecure channel and derive a shared secret key.
- The Diffie-Hellman key exchange uses a large cyclic group and is secure against eavesdropping under the decisional Diffie-Hellman assumption. The plain Diffie-Hellman protocol does not protect against active attacks.
- A standard choice for Diffie-Hellman are subgroups of the multiplicative group of integers modulo a prime number. The subgroup should have a large prime order.
- Diffie-Hellman can be broken by computing discrete logarithms, but this is assumed to be a hard problem if the group is appropriately chosen.
- Key encapsulation mechanisms are based on a public-key scheme, for example RSA or Diffie-Hellman. A secret key is encapsulated using the public key of the recipient.
- Hybrid public-key encryption schemes combine key encapsulation, symmetric encryption and message authentication. Such schemes can have a security guarantee and can be used to protect large data sets.

Exercises

1. Show that the discrete-logarithm problem is easy in the additive group $(\mathbb{Z}_p, +)$.
2. How can you efficiently generate Diffie-Hellman parameters p, q and g for the multiplicative group $\mathbb{Z}_p^*$ with given bit lengths n_p and n_q for p and q?
3. Let $p = 89$, $g = 2 \mod 89$ and $G = \langle g \rangle$. How many different shared keys k are possible in a Diffie-Hellman key exchange with these parameters?
4. You perform a Diffie-Hellman key exchange with Alice and you agreed on the parameters $p = 43$, $G = \langle g \rangle \subset \mathbb{Z}_p^*$ and $g = 3 \mod 43$.
   (a) Determine $q = \text{ord}(g)$.
   (b) Alice sends you $A = 14$ and you choose the secret exponent $b = 26$. Which value do you send to Alice? Compute the shared secret key k.
5. Let $g \equiv 3$ be an element of the group $\mathbb{Z}_{107}^*$.
   (a) Show that g generates a group G of prime order.
   (b) How many exponentiations at most are necessary to compute a discrete logarithm in G using the Babystep-Giantstep algorithm?
   (c) Compute $\log_3(12)$ in G.

6. Show that the following parameters (a 2048-bit MODP group given in RFC 5114 [**LK08**]) can be used in a Diffie-Hellman key exchange, i.e., show that p and q are prime numbers and $\text{ord}(g) = q$.
   Tip: Use SageMath. Remove the line breaks and define strings. The corresponding hexadecimal numbers can be constructed with `ZZ( ..., 16)`. Use the function `is_pseudoprime( )` to check the primality.

```
p = 87A8E61D B4B6663C FFBBD19C 65195999 8CEEF608 660DD0F2
    5D2CEED4 435E3B00 E00DF8F1 D61957D4 FAF7DF45 61B2AA30
    16C3D911 34096FAA 3BF4296D 830E9A7C 209E0C64 97517ABD
    5A8A9D30 6BCF67ED 91F9E672 5B4758C0 22E0B1EF 4275BF7B
    6C5BFC11 D45F9088 B941F54E B1E59BB8 BC39A0BF 12307F5C
    4FDB70C5 81B23F76 B63ACAE1 CAA6B790 2D525267 35488A0E
    F13C6D9A 51BFA4AB 3AD83477 96524D8E F6A167B5 A41825D9
    67E144E5 14056425 1CCACB83 E6B486F6 B3CA3F79 71506026
    C0B857F6 89962856 DED4010A BD0BE621 C3A3960A 54E710C3
    75F26375 D7014103 A4B54330 C198AF12 6116D227 6E11715F
    693877FA D7EF09CA DB094AE9 1E1A1597
g = 3FB32C9B 73134D0B 2E775066 60EDBD48 4CA7B18F 21EF2054
    07F4793A 1A0BA125 10DBC150 77BE463F FF4FED4A AC0BB555
    BE3A6C1B 0C6B47B1 BC3773BF 7E8C6F62 901228F8 C28CBB18
    A55AE313 41000A65 0196F931 C77A57F2 DDF463E5 E9EC144B
    777DE62A AAB8A862 8AC376D2 82D6ED38 64E67982 428EBC83
    1D14348F 6F2F9193 B5045AF2 767164E1 DFC967C1 FB3F2E55
    A4BD1BFF E83B9C80 D052B985 D182EA0A DB2A3B73 13D3FE14
    C8484B1E 052588B9 B7D2BBD2 DF016199 ECD06E15 57CD0915
    B3353BBB 64E0EC37 7FD02837 0DF92B52 C7891428 CDC67EB6
    184B523D 1DB246C3 2F630784 90F00EF8 D647D148 D4795451
    5E2327CF EF98C582 664B4C0F 6CC41659
q = 8CF83642 A709A097 B4479976 40129DA2 99B1A47D 1EB3750B
    A308B0FE 64F5FBD3
```

7. Let $p = 59$, $g \equiv 4 \in \mathbb{Z}_p^*$, $q = \text{ord}(g) = 29$, $G = \langle g \rangle$ and $A = g^a \equiv 9$. Compute the discrete logarithm $a = \log_g(A)$ in G with Pollard's ρ method.
   Hint: Use the collision $g^2 A^1 = g^5 A^5$ in G.
8. Show that $g = 11$ generates the group $G = \mathbb{Z}_{109}^*$. Apply the Pohlig-Hellman algorithm to compute the discrete logarithm $\log_{11}(54)$.
9. Why is RSA key encapsulation (see Definition 10.14) not CPA-secure without the hashing operation?
10. Discuss the consequences of re-using one or both of the secret Diffie-Hellman keys a and b.
11. Explain a Man-in-the-Middle attack against the Diffie-Hellman protocol.
12. The *ElGamal public-key encryption scheme* uses the same parameters as the Diffie-Hellman key-exchange, i.e., a cyclic group G, a generator g and $q = \text{ord}(g)$. Choose a uniform number $a \in \mathbb{Z}_q$ and set $A = g^a \in G$. The message space is G, the ciphertext space is $G \times G$, the public key is $pk = (G, q, g, A)$ and the private key

is $sk = (G, q, g, a)$. Encryption is randomized; to encrypt $m \in G$, one chooses a uniform number $b \in \mathbb{Z}_q$ and defines the ciphertext as

$$\mathcal{E}_{pk}(m) = (g^b, A^b m).$$

Decryption is given by

$$\mathcal{D}_{sk}(c_1, c_2) = c_1^{-a} c_2.$$

(a) Show that the scheme provides correct decryption.

(b) Assume that $p = 59$, $G = \mathbb{Z}_p^*$, $g \equiv 4$ and $a = 20$ define Alice's ElGamal key. She obtains the ciphertext $c = (17, 16)$. Compute the plaintext m.

(c) The ElGamal encryption is randomized by a parameter b. Explain why b has to remain secret and should not be re-used for different encryptions.

Chapter 11

Digital Signatures

Digital signatures are asymmetric cryptographic schemes which aim at data *integrity and authenticity*. There are some similarities to message authentication codes, but digital signatures are verified using a *public key*. Successful verification shows that the data is authentic and has not been tampered with. Since the private key is exclusively controlled by the signer, digital signatures achieve not only data integrity and authenticity, but also *non-repudiation*. Signatures have applications beyond integrity protection, for example in entity authentication protocols.

In Section 11.1, we define digital signature schemes and discuss their security: signatures should be unforgeable. Section 11.2 deals with the definition of the plain RSA signature, which is based on the same parameters as the RSA cryptosystem. The plain RSA signature is forgeable, and hashing of the data is advisable for security and efficiency reasons. Furthermore, we present the probabilistic RSA-PSS scheme in Section 11.3. Other signatures schemes (ElGamal and DSA) are briefly discussed in the exercises at the end of this chapter.

We refer the reader to [**PP10**], [**KL15**] and [**GB08**] for additional reading.

11.1. Definitions and Security Requirements

Signature schemes consist of key generation, signing and a verification algorithm.

Definition 11.1. A *digital signature* scheme is given by the following spaces and polynomial-time algorithms:

- A message space $\mathcal{M}$.
- A space of key pairs $\mathcal{K} = \mathcal{K}_{pk} \times \mathcal{K}_{sk}$.

- A randomized key generation algorithm $Gen(1^n)$ that takes a security parameter 1^n as input and outputs a pair of keys (pk, sk).
- A signing algorithm, which may be randomized. It takes a message m and a private key sk as input and outputs a signature $s \leftarrow sign_{sk}(m)$.
- A deterministic verification algorithm that takes a public key pk, a message m and a signature s as input and outputs 1 if the signature is valid, and 0 otherwise. ◊

Note that verification of a signature also requires the message. The signature is usually short and does not include the message.

Similar to a message authentication code, the security of a signature scheme is determined by the hardness of computing a valid signature without the private key. We assume that an adversary knows the public key and is thus able to verify signatures. Furthermore, we assume that the adversary can choose arbitrary messages to be signed. This is called a *chosen message attack* and corresponds to a situation in practice where many signature values are known and legitimate or innocent messages are routinely signed.

Definition 11.2. Suppose a signature scheme is given. Consider the following experiment (see Figure 11.1): a challenger takes the security parameter 1^n as input and generates a key pair (pk, sk) by running $Gen(1^n)$. An adversary A is given 1^n and the public key pk. The adversary can choose messages m and obtains the signature $s = sign_{sk}(m)$ from an oracle. A can also verify signatures using the public key pk. The adversary tries to forge a signature of a new message m' and outputs (m', s'). The challenger outputs 1 if the signature is valid and has not been queried before, and 0 otherwise.

The scheme is called *existentially unforgeable under an adaptive chosen message attack* (*EUF-CMA secure* or just *secure*) if for all probabilistic polynomial-time adversaries, the probability of success is negligible in n. ◊

The definition of a secure scheme requires that the length of signature values is not too short, since an adversary might otherwise guess valid signatures.

Digital signatures protect the integrity and authenticity of messages and can also achieve *non-repudiation*. Since the signer alone controls the private key, they cannot deny the signature afterwards. Note that symmetric schemes cannot achieve non-repudiation, since the secret key is known to two (or more) parties.

Remark 11.3. The verification of a digital signature requires the *authentic public key* of the signer. Although public keys can be openly shared, authenticity is not evident. A *man-in-the-middle* might replace the message, the signature and the public key with his own data. A verifier is not able to detect this attack, unless they can check the authenticity of the public key. In practice, public keys are often signed by other parties (in a *Web of Trust*) or by a *Certification Authority* (CA). A signed *certificate* binds the

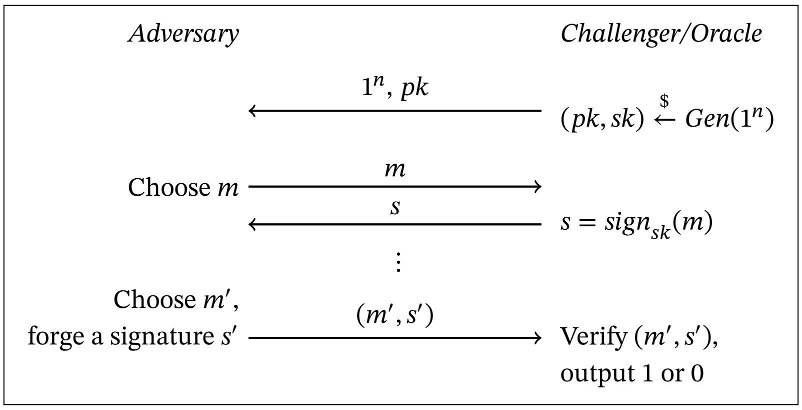

Adversary
Challenger/Oracle
$1^n, pk$
$(pk, sk) \overset{\$}{\leftarrow} Gen(1^n)$
Choose m
m
s
$s = sign_{sk}(m)$
Choose m', forge a signature s'
(m', s')
Verify (m', s'), output 1 or 0

Figure 11.1. Signature forgery experiment.

identity of a subject to its public key. This shifts the problem of authentic public keys to a third party that is hopefully trustworthy.

11.2. Plain RSA Signature

The plain RSA signature scheme is defined analogously to the plain RSA encryption scheme (see Chapter 9.2). Signing corresponds to decryption and requires the private RSA key; verification corresponds to encryption and uses the public RSA key.

Definition 11.4. The plain RSA signature scheme is defined by:

- A key generation algorithm $Gen(1^n)$ that takes a security parameter 1^n as input, generates two random primes p and q of length n and sets $N = pq$. Furthermore, integers e and d with
$$ed \equiv 1 \mod (p-1)(q-1)$$
are chosen as in the RSA encryption scheme. $Gen(1^n)$ outputs the public key $pk = (e, N)$ and the private key $sk = (d, N)$.
- The message space is $\mathcal{M} = \mathbb{Z}_N^*$.
- The deterministic signature algorithm takes the secret key $sk = (d, N)$ and a message $m \in \mathcal{M}$ as input and outputs the signature
$$s = sign_{sk}(m) = m^d \mod N.$$
- The verification algorithm takes the public key $pk = (e, N)$, a message $m \in \mathbb{Z}_N$ and a signature s. It computes
$$s^e \mod N$$
and outputs 1 (signature is valid) if $m = s^e \mod N$, and 0 otherwise. $\Diamond$

The correctness follows in the same way as for the RSA encryption algorithm.

The efficiency of RSA was discussed in Section 9.5. The complexity of signature verification is $O(n^2)$ if the public exponent e is short, for example $e = 2^{16} + 1$. The running time of the RSA signature is $O(n^3)$ because the size of the private exponent d is n. The exponentiation can be accelerated by a factor of around 4 using the Chinese Remainder Theorem (compare Example 9.13). Digital signatures are not as efficient as message authentication codes (see Chapter 8), and one avoids carrying out a large number of signatures or verifications.

Example 11.5. Alice's RSA parameters are $p = 11$, $q = 23$, $N = pq = 253$, $e = 3$ and $d = 147$. She signs the message $m = 111$ and computes $s = 111^{147} \mod 253 \equiv 89$. Bob uses her public key $pk = (3, 253)$ and verifies the signature by computing $89^3 \mod 253 \equiv 111$. ♢

Unfortunately, this scheme is both impractical and insecure. Firstly, the message length is limited by the size of the RSA modulus N, but in practice, one needs to sign messages of *arbitrary length* and not only several hundred bytes, the usual RSA modulus length.

Furthermore, the plain RSA signature scheme is insecure, because the signature is *multiplicative* and new signature values can be easily forged. If s_1 and s_2 are signatures of m_1 and m_2, then $s_1 s_2 \mod N$ is a valid signature of $m_1 m_2 \mod N$. Similarly, valid signature values can be generated by taking powers. An adversary can even choose any $s \in \mathbb{Z}_N^*$ and compute $m = s^e \mod N$. Then s is a valid signature of the message m. This attack is called *existential forgery*. Note that the adversary only controls the signature value s and not the message m.

Example 11.6. Assume Alice's RSA parameters are the same as in Example 11.5 above. Mallory generates a forged signature value $s = 123$ and computes $m = s^3 \equiv 52 \mod 253$. Bob successfully verifies the signature s of m:

$$s^e = 123^3 \equiv 52 \mod 253.$$

But Alice has never signed $m = 52$. ♢

We conclude that the plain RSA signature scheme is *not EUF-CMA secure*. This is analogous to the fact that plain RSA encryption is malleable and insecure under a chosen ciphertext attack (see Chapter 9.2).

11.3. Probabilistic Signature Scheme

An obvious improvement of the plain RSA signature scheme is to apply a *hash function* before signing a message. If the output length of the hash function is at most the length of the RSA modulus, then messages of arbitrary length can be signed. Furthermore, hashing prevents attacks that exploit the multiplicative structure of the plain RSA scheme.

Closer analysis shows that the hash function should be collision-resistant and have the properties of a random oracle. Furthermore, the range of the hash function should be the full RSA message space $\mathbb{Z}_N^*$. The latter poses a problem in practice, since the RSA modulus is usually more than 2000 bits long, whereas the digests of well-known hash functions are much shorter – only between 160 and 512 bits.

The RSA-FDH (*Full Domain Hash*) signature is similar to the plain RSA scheme, but leverages a hash function $H : \{0,1\}^* \to \mathbb{Z}_N^*$. A message m is first hashed and then signed:

$$s = sign_{sk}(m) = H(m)^d \mod N.$$

In the verification step, $H(m)$ is computed and then compared to $s^e \mod N$. The signature is valid if $H(m) = s^e \mod N$.

Obviously, the collision-resistance of H is crucial, since a collision $H(m_1) = H(m_2)$ with $m_1 \neq m_2$ can be used for an existential forgery.

Theorem 11.7. *If H has range $\mathbb{Z}_N^*$ and is modeled as a random oracle, the RSA-FDH scheme is EUF-CMA secure under the RSA assumption.*

Remark 11.8. The above theorem has a proof by reduction (see [**BR05**], [**KL15**]). If we assume that the hash function behaves like a random oracle, then forging a signature is only possible by inverting the RSA function $f(x) = x^e \mod N$ on uniform random integers modulo N. But under the RSA assumption, the probability of successfully inverting f is negligible. ◊

Since the length of cryptographic hashes is usually smaller than the size of the RSA modulus, one stretches the hash by randomized padding. The result should still be indistinguishable from a random integer in $\mathbb{Z}_N$. A standard method is defined in *PKCS #1 version 2.2* (see RFC 8017 [**MKJR16**]) and is called a *Probabilistic Signature Standard* (RSA-PSS) or RSASSA-PSS (RSA signature scheme with appendix).

Similar to RSA-OAEP (see Chapter 9.6), the PSS encoding requires a hash function H with output byte length h and a mask generating function MGF with input length h and variable output length. For example, H could be SHA-2 and MGF is based on this hash function.

In the following, we describe RSA-PSS signing and verification (see Figure 11.2). Let m be the message. We will derive an encoded message EM of byte length $k = \lceil \frac{\text{size}(N)-1}{8} \rceil$ and sign EM.

First, m' is defined by concatenating eight zero padding bytes, the hashed message $H(m)$ and a random *salt* string. A typical salt length is h bytes, but an empty *salt* is also permitted:

$$m' = \texttt{00}^8 \,||\, H(m) \,||\, salt.$$

The result is hashed again and we obtain $H(m')$. A data block DB of length $k - h - 1$ is formed by concatenating the necessary number of zero padding bytes, one byte 01 and the *salt* string:

$$DB = \texttt{00} \| \ldots \| \texttt{00} \| \texttt{01} \| \ salt.$$

Then $MGF(H(m'), k-h-1)$ outputs a string of the same length as DB. Set

$$maskedDB = DB \oplus MGF(H(m'),\ k-h-1)$$

and define the encoded message EM by concatenating $maskedDB$, the hash $H(m')$ and the byte `BC`:

$$EM = maskedDB \ \| \ H(m') \ \| \ \texttt{BC}.$$

The byte length of EM is k. Finally, the signature is defined by

$$s = sign_{sk}(m) = EM^d \mod N.$$

The verification involves various format and length checks. Compute

$$EM = s^e \mod N.$$

The rightmost byte of EM should be `BC`. Then the byte strings $maskedDB$ of length $k-h-1$ and H' of length h are extracted and

$$DB = maskedDB \oplus MGF(H',\ k-h-1)$$

is computed. The leftmost bytes of DB should be `00`, followed by `01`. The remaining bytes of DB form the *salt*. Set

$$m' = \texttt{00}^8 \ \| \ H(m) \ \| \ salt$$

and compute $H(m')$. If $H(m') = H'$ then the signature is valid. Otherwise, the signature is invalid. It is important that only one failure message is given for all format or verification errors.

Remark 11.9. If the salt is randomly chosen and is sufficiently long, then the RSA-PSS signature is randomized and signing the same message twice using the same key will give different signature values. An adversary who compares different signature values cannot see whether any of the underlying messages are identical. ♢

The following Theorem on the security of RSA-PSS has a proof by reduction ([**BR96**], [**BR05**]). However, the original RSA-PSS scheme is not identical to the standardized version.

Theorem 11.10. *The RSA-PSS signature scheme is EUF-CMA secure in the random oracle model under the RSA assumption.* ♢

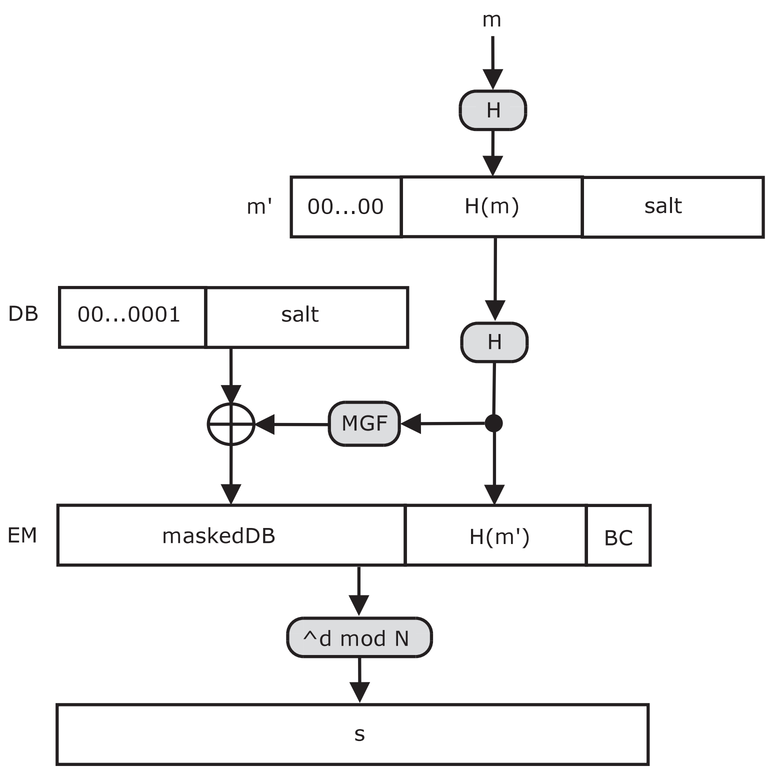


Figure 11.2. Signing a message *m* using RSA-PSS.

So far we have only discussed RSA signatures. An alternative is signature schemes which are based on the discrete logarithm problem in a cyclic group. In particular, the units of a finite field and the points of an elliptic curve can be used, and the corresponding signature schemes are ElGamal, DSA/DSS (Digital Signature Algorithm) and ECDSA (Elliptic Curve Digital Signature Algorithm). ElGamal and the standardized scheme DSA are explained in Exercise 10 and elliptic curves are explored in Chapter 12.

Signatures are also used in authenticated key exchange (AKE) and in entity authentication protocols. If the public key is authentic, then a correct signature shows that the signer controls the corresponding private key. This can be used as a proof of identity.

Another interesting topic not covered in this book is *hash-based signatures*. In 1975, Lamport discovered a one-time signature system based on hashes without the use of number-theoretic constructions [**Lam79**], and various improvements of the original system exist. Hash-based signatures are also a promising candidate for post-quantum cryptography.

11.4. Summary

- Digital signature schemes are based on public-key cryptography. The private key is used to sign a message and the public key to verify a signature.
- Digital signatures can achieve integrity, authenticity and non-repudiation of messages.
- The RSA signature algorithm uses the same parameters as the RSA cryptosystem.
- The plain RSA signature scheme has serious weaknesses and the input data should at least be hashed.
- RSA-PSS is a standardized signature scheme that applies hashing and padding before signing a message using RSA.
- The Digital Signature Algorithm (DSA/DSS) is a standardized signature scheme that is based on the discrete logarithm problem.

Exercises

1. Give possible reasons why a signature may be invalid.
2. Suppose a new signature scheme takes pairs (m_1, m_2) of messages of the same length as input and defines the signature as follows: set $m = m_1 \oplus m_2$ and set $s = sign_{sk}(m)$ using a secure signature scheme. Is the new scheme secure or would you rather change this scheme?
3. Let $e = 5$, $N = 437$ be the parameters of a public RSA key. Verify the following plain RSA signatures:
   (a) $m = 102, s = 416$,
   (b) $m = 101, s = 416$,
   (c) $m = 100, s = 86$.
   Conduct an existential forgery attack using the signature value $s = 99$.
4. Compute the RSA-FDH signature of a message m with hash value $H(m) = 11111$ using the RSA parameters $N = 28829$ and $e = 5$, where d has to be determined. Verify the signature with the public RSA key. *Hint:* $s = 7003$.
5. Consider the verification of an RSA-FDH signature. It is sufficient to give only the hash and the signature value to a verifier and not the message?
6. Why does the existential forgery attack almost certainly fail for RSA-PSS?
7. Discuss the consequences of a fixed and known salt value in RSA-PSS.
8. Show that the plain RSA signature scheme is secure under the RSA assumption if one considers the following experiment: the adversary succeeds if they output a valid signature of a challenge message which is chosen uniformly at random.

9. Suppose we want to sign and to encrypt data (*signcryption*). Analogue to the encrypt-then-authenticate approach in symmetric cryptography, Alice encrypts a secret message with Bob's public key and then signs the ciphertext using her private key.
   (a) How can an adversary produce their own signature of the same message and thus mislead Bob?
   (b) Does this combination of encryption and signature provide non-repudiation?
10. The *ElGamal signature scheme* is based on the discrete logarithm problem and uses a cyclic subgroup G of $\mathbb{Z}_p^*$ of order q and a generator $g \in G$. One chooses a secret uniform $a \in \mathbb{Z}_q$ and computes $A = g^a$. Then $pk = (p, g, q, A)$ forms the public key and $sk = (p, g, q, a)$ is the secret key. The signature generation is randomized; one chooses a random uniform $k \in \mathbb{Z}_q^*$ and computes the signature value by
$$sign_{sk}(m) = (r, s) \text{ with } r \equiv g^k \mod p \text{ and } s \equiv k^{-1}(H(m) - ar) \mod q.$$
To verify the signature (r, s) of a message m, one computes $A^r\, r^s \mod p$ and compares the result with $g^{H(m)} \mod p$. If both residue classes coincide, then the signature is valid.

   The *Digital Signature Algorithm* (called DSA or DSS) is a standardized variant of the ElGamal signature scheme [**FIP13**]. The bit lengths of p and q are specified, e.g., size(p) = 2048 and size(q) = 224. Unlike ElGamal, both DSA signature parts r and s are reduced modulo q and the verification is also performed modulo q. Although q is much smaller than p, the existing sub-exponential attacks against the discrete logarithm problem do not run faster in subgroups of $\mathbb{Z}_p^*$. Other attacks, e.g., Babystep-Giantstep and Pollard's ρ-algorithm, can use the subgroup, but their running time is $O(2^{\text{size}(q)/2})$ and thus out of reach if q has more than 200 bits.
   (a) Show that the verification is correct.
   (b) Assume that $p = 59$, $g \equiv 4$, $q = 29$ and $a = 20$ form Alice's ElGamal key. Compute the public parameter A.
   (c) Alice wants to sign a message m with hash value $H(m) = 8$. She chooses the secret parameter $k = 5$. Compute the ElGamal signature (r, s).
   (d) Check that the signature (r, s) is valid.
   (e) The ElGamal signature is randomized by k. Explain why k must remain secret and should not be re-used for different signatures.

Chapter 12

Elliptic Curve Cryptography

In this chapter, we outline the basics of Elliptic Curve Cryptography (ECC). In several public-key schemes, the additive group of points on elliptic curves over finite fields forms an alternative to the multiplicative group of integers modulo a prime number. The addition of points on an elliptic curve is slightly more complex than the multiplication of residue classes, but elliptic curves offer a similar level of security as the multiplicative group with shorter keys. ECC is now widely used because of its efficiency and accepted security.

In Section 12.1, Weierstrass equations and elliptic curves are introduced and the addition of points on a cubic curve is explained. We present the Elliptic Curve Diffie-Hellman algorithm in Section 12.2 and discuss the efficiency and security of ECC in Section 12.3. Finally, in Section 12.4 we show how elliptic curves can be leveraged to factor integers. The Elliptic Curve Digital Signature Algorithm (ECDSA) is discussed in the exercises at the end of this chapter.

Elliptic curves over different fields are an interesting and challenging mathematical topic and we refer to [**Sil09**] for an in-depth treatment. There are several textbooks on cryptographic applications of elliptic curves and we recommend [**Was08**], [**TW06**], [**Gal12**] and [**She17**] for additional reading.

12.1. Weierstrass Equations and Elliptic Curves

The multiplicative group $\mathbb{Z}_p^*$ of integers modulo a prime number p is used in several cryptographic algorithms, for example in the Diffie-Hellman key exchange. A well-proven alternative is the group of points on an *elliptic curve* over a finite field. Elliptic curves are defined by *Weierstrass equations*.

Definition 12.1. Let K be a field and $a_1, \dots, a_6 \in K$. Then the equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

is called a (generalized) *Weierstrass equation*. The equation defines a cubic curve called a *Weierstrass curve*. ◊

In many cases, the generalized Weierstrass equation can be transformed into a shorter equation.

Definition 12.2. Let K be a field and $a, b \in K$. Then the equation

$$y^2 = x^3 + ax + b$$

is called a *short Weierstrass equation*.

Proposition 12.3. *Let K be a field with char$(K) \neq 2, 3$. Then a Weierstrass curve over K is up to an affine transformation given by a short Weierstrass equation.*

Proof. By assumption, we can divide by 2 and 3. Following [**Was08**], we write the generalized Weierstrass equation as

$$\left(y + \frac{a_1}{2}x + \frac{a_3}{2}\right)^2 = x^3 + \left(a_2 + \frac{a_1^2}{4}\right)x^2 + \left(a_4 + \frac{a_1a_3}{2}\right)x + \left(\frac{a_3^2}{4} + a_6\right).$$

The affine transformation $y' = y + \frac{a_1}{2}x + \frac{a_3}{2}$ yields an equation

$$(y')^2 = x^3 + a_2'x^2 + a_4'x + a_6'$$

with constants $a_2', a_4', a_6' \in K$. Finally, replacing x by $x' - \frac{a_2'}{3}$ removes the x^2 term and gives the desired equation

$$(y')^2 = (x')^3 + ax' + b$$

for some constants $a, b \in K$. □

From now on, we assume that *char*(K) is neither 2 nor 3, although this excludes the binary fields $K = GF(2^m)$. The theory of elliptic curves also works over these fields and are used in cryptography, but there are some technical differences, for example with respect to the Weierstrass equation.

We want to add *points at infinity* to the n-dimensional space K^n.

Definition 12.4. Let K be a field and $n \in \mathbb{N}$. Then the n-dimensional *projective space* $\mathbb{P}^n(K)$ is defined as the set of all lines in K^{n+1} passing through the origin. Points in $\mathbb{P}^n(K)$ are given by $n + 1$ *projective* or *homogeneous* coordinates and are denoted by

$$[x_1 : x_2 : \cdots : x_{n+1}].$$

Two points are equivalent and give the same element in $\mathbb{P}^n(K)$ if they are on the same line, i.e., if they differ only by a nonzero factor $\lambda \in K$. Hence the projective space $\mathbb{P}^n(K)$ is a set of equivalence classes of $K^{n+1} \setminus \{0^{n+1}\}$, where

$$[x_1 : x_2 : \cdots : x_{n+1}] \sim [y_1 : y_2 : \cdots : y_{n+1}]$$

if there exists a $\lambda \in K^*$ such that $y_1 = \lambda x_1, y_2 = \lambda x_2, \dots, y_{n+1} = \lambda x_{n+1}$. ◊

The points $[x_1 : \cdots : x_n : 0]$ are said to be *points at infinity*. The usual space K^n of n-dimensional vectors is called the *affine space* and we will sometimes write $\mathbb{A}^n(K)$ instead of K^n. We have an injection

$$\mathbb{A}^n(K) \hookrightarrow \mathbb{P}^n(K),\ (x_1, \dots x_n) \mapsto [x_1 : \cdots : x_n : 1]$$

and the complement of the image under this map consists of the points at infinity.

Example 12.5. a) Points in $\mathbb{P}^1(K)$ are lines in the plane K^2 passing through the origin. If $y \neq 0$ then $[x : y]$ is equivalent to $[\frac{x}{y} : 1]$, which lies in the image of $\mathbb{A}^1(K)$. Otherwise, $[x : 0] \sim [1 : 0]$ is the point at infinity. Hence

$$\mathbb{P}^1(K) = \mathbb{A}^1(K) \cup \{[1 : 0]\}.$$

b) In the two-dimensional projective space $\mathbb{P}^2(K)$, all points are either equivalent to $[x : y : 1]$ or to $[x : y : 0]$. We have a decomposition

$$\mathbb{P}^2(K) = \mathbb{A}^2(K) \cup \mathbb{A}^1(K) \cup \{[1 : 0 : 0]\},$$

where $(x, y) \in K^2$ corresponds to $[x : y : 1]$ and $u \in K$ to the point $[u : 1 : 0]$. ◊

A curve in the affine space $\mathbb{A}^2(K)$ can be extended to a *projective curve* in $\mathbb{P}^2(K)$. Suppose the curve is given by the Weierstrass equation $y^2 = x^3 + ax + b$. We set $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$ and obtain

$$\frac{Y^2}{Z^2} = \frac{X^3}{Z^3} + a\frac{X}{Z} + b.$$

Multiplying both sides by Z^3 yields the Weierstrass equation in projective (or homogeneous) coordinates:

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

Proposition 12.6. *The points on the projective curve $Y^2Z = X^3 + aXZ^2 + bZ^3$ are either equivalent to $[x : y : 1]$, where $(x, y) \in K^2$ satisfies the affine Weierstrass equation $y^2 = x^3 + ax + b$, or to the point $[0 : 1 : 0]$ at infinity.*

Proof. We only need to consider points $[X : Y : 0]$ on the projective Weierstrass curve, but $Z = 0$ implies $X^3 = 0$ and thus $X = 0$. The only remaining point is $[0 : 1 : 0]$, which in turn satisfies the Weierstrass equation. □

We denote the point $[0 : 1 : 0]$ at infinity by O. This point lies on every vertical line $x = c$, since the corresponding equation in projective coordinates is

$$X = cZ$$

and $O = [0 : 1 : 0]$ satisfies this equation for every $c \in K$.

Definition 12.7. Let C be an affine curve given by a polynomial equation

$$f(x, y) = 0$$

over a field K. Let $\overline{K}$ be the algebraic closure of K (see Remark 4.72). Then C is called *nonsingular* (or smooth) if for all points $P = (x, y) \in C(\overline{K})$ the partial derivatives $D_x f$ and $D_y f$ do not simultaneously vanish at P:

$$((D_x f)(P), (D_y f)(P)) \neq (0, 0).$$

$D_x f = D_x(f)$ and $D_y f = D_y(f)$ are the formal derivatives of f with respect to x and y, respectively (see Definition 4.55). If both derivatives vanish at P, then P is called a *singular* point. A point P on the corresponding projective curve $f(X, Y, Z) = 0$ is nonsingular if

$$((D_X f)(P), (D_Y f)(P), (D_Z f)(P)) \neq (0, 0, 0).$$

Example 12.8. Let C be the Weierstrass curve $y^2 = x^3$ over a field K. Then $f(x, y) = -y^2 + x^3$ and

$$D_x f = 3x^2 \text{ and } D_y f = -2y.$$

We assumed that 2 and 3 are nonzero in K. Then the equations $3x^2 = 0$ and $-2y = 0$ give $(x, y) = (0, 0)$, a point on the curve. Therefore, C is singular at the point $(0, 0)$ and nonsingular at all other points, including the point $O = [0 : 1 : 0]$ at infinity, since $f(X, Y, Z) = -Y^2 Z + X^3$ gives $D_Z f = -Y^2$ and $(D_Z f)(O) = -1$. ♢

We want to give a condition under which a Weierstrass curve is nonsingular.

Proposition 12.9. *Let $y^2 = x^3 + ax + b$ be a Weierstrass curve over a field K. Then*

$$\Delta = -16(4a^3 + 27b^2)$$

is called the discriminant *of the curve. If Δ is nonzero in K, then the curve is nonsingular.*

Proof. The curve is defined by the equation $f(x, y) = -y^2 + x^3 + ax + b = 0$. One has

$$D_x f = 3x^2 + a \text{ and } D_y f = -2y.$$

Suppose that $(x, y) \in \overline{K} \times \overline{K}$ lies on the Weierstrass curve and both formal partial derivatives vanish; then $y = 0$ and $a = -3x^2$. Since $f(x, y) = 0$ we also have $x^3 + ax + b = 0$, which gives $x^3 - 3x^3 + b = 0$ and $b = 2x^3$. The equations for $a = -3x^2$ and $b = 2x^3$ imply

$$-4a^3 = 27b^2 = 108x^6.$$

If $\Delta \neq 0$ then $-4a^3 \neq 27b^2$, and so the affine curve does not have singular points.

It remains to be shown that the point $O = [0 : 1 : 0]$ at infinity is also nonsingular. In projective coordinates, we have

$$f(X, Y, Z) = -Y^2 Z + X^3 + aXZ^2 + bZ^3.$$

The partial derivative with respect to Z is

$$D_Z f = -Y^2 + 2aXZ + 3bZ^2,$$

and thus $(D_Z f)(O) = -1$, which shows that O is a nonsingular point. □

The above proof also shows that a curve defined by a short Weierstrass equation is nonsingular if and only if the cubic $x^3 + ax + b$ does not have a double root.

Definition 12.10. An *elliptic curve* E over a field K is a nonsingular projective curve defined by a Weierstrass equation. The set of points on E with coordinates in K is denoted by $E(K)$. $\Diamond$

We have seen that $E(K) \subset \mathbb{P}^2(K)$ consists of all points satisfying the affine Weierstrass equation and one additional point $O = [0 : 1 : 0]$ at infinity, i.e.,

$$E(K) = \{(x, y) \in K \times K \mid y^2 = x^3 + ax + b\} \cup \{O\}.$$

A very important fact is that points in $E(K)$ *can be added*. However, addition is not the usual vector addition in K^2, since the sum would not lie on the curve. Instead, we will show that a line through two nonsingular points P and Q intersects the elliptic curve at a third point R (see Figure 12.1), and use this property to define the addition. Let E be an elliptic curve defined by the equation $f(x, y) = -y^2 + x^3 + ax + b = 0$. It is necessary to consider different cases:

(1) Firstly, we suppose that $P = (x_1, y_1)$, $Q = (x_2, y_2) \in E(K)$ are points with $P,\ Q \neq O$ and different x-coordinates. Then a straight line through P and Q intersects the Weierstrass curve of degree 3 at a third point $R = (x_3, y_3) \in E(K)$. In fact, the equation of the line through P and Q is

$$y = l(x) = m(x - x_1) + y_1, \text{ where } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

In order to find the x-coordinate of the third intersection point, we replace y by $l(x)$ in the Weierstrass equation $-y^2 + x^3 + ax + b = 0$. This gives a cubic equation in the variable x:

$$f(x, l(x)) = -(m(x - x_1) + y_1)^2 + x^3 + ax + b = x^3 - mx^2 + \cdots = 0.$$

Since this equation already has two zeros in K (the x-coordinates of P and Q), it must have a third zero $x_3 \in K$, and hence the cubic polynomial can be factorized:

$$f(x, l(x)) = (x - x_1)(x - x_2)(x - x_3) = x^3 - (x_1 + x_2 + x_3)x^2 + \dots .$$

By comparing the x^2-terms in the above expressions, we find that

$$x_1 + x_2 + x_3 = m^2.$$

Thus the coordinates of $R = (x_3, y_3)$ are

$$x_3 = m^2 - x_1 - x_2 \text{ and } y_3 = m(x_3 - x_1) + y_1.$$

(2) If $P,\ Q \in E(K)$ with $P,\ Q \neq O$ have the same x-coordinate, then the line through P and Q is a vertical line and, as we saw above, the point O at infinity lies on all vertical lines. Hence the third point is $R = O$.

(3) Now let $P = Q = (x_1, y_1) \in E(K)$ and $P \neq O$. If $y_1 \neq 0$ then we take the tangent line at P. The tangent at P is the line of equation

$$(x - x_1)(D_x f)(P) + (y - y_1)(D_y f)(P) = 0,$$

and rearranging gives the equation

$$y = t(x) = m(x - x_1) + y_1 \text{ , where } m = -\frac{(D_x f)(P)}{(D_y f)(P)} = \frac{3x_1^2 + a}{2y_1}.$$

Replacing y with $t(x)$ in the Weierstrass equation $f(x,y) = -y^2 + x^3 + ax + b = 0$ gives a cubic equation with a double root at x_1. We denote the other root by x_3. This is the x-coordinate of the point $R = (x_3, y_3)$, the intersection of the tangent with the elliptic curve. Factorization of the cubic polynomial gives

$$f(x, t(x)) = (x - x_1)^2(x - x_3) = x^3 - (x_3 + 2x_1)x^2 + \dots .$$

On the other hand, we have as above $f(x, t(x)) = x^3 - m^2x^2 + \dots$, and comparing the x^2-terms yields

$$x_3 = m^2 - 2x_1.$$

The corresponding y-coordinate of R is

$$y_3 = m(x_3 - x_1) + y_1.$$

We obtain almost the same formulas as in the case $P \neq Q$, but the slope m is defined differently.

(4) If $P = Q = (x_1, y_1) \in E(K)$ and $y_1 = 0$, then $x = x_1$ is a vertical tangent line and $R = O$ lies on that line.

(5) If $P = O$ and $Q = (x_1, y_1) \in E(K)$, then the line through P and Q is the vertical line $x = x_1$, which intersects the elliptic curve in $R = (x_1, -y_1)$. Accordingly, if $P = (x_1, y_1)$ and $Q = O$ then $R = (x_1, -y_1)$.

(6) Finally, if $P = Q = O$ then $R = O$.

In summary, given two points $P, Q \in E(K)$, there is a unique third point $R \in E(K)$ such that P, Q and R (with multiplicities) lie on the same line. We can therefore define the addition of points on E by letting

$$P + Q + R = O \Longleftrightarrow P + Q = -R$$

(see Figure 12.1). $-R$ is the reflection of $R = (x_1, y_1)$ across the x-axis, i.e.,

$$-R = (x_1, -y_1).$$

Note that the line through R and $-R$ is the vertical line $x = x_1$ and the third point on that line is O, so that $R + (-R) + O = O$, as expected.

In our computations above, we derived explicit formulas for the inverse point and the addition of two points:

Proposition 12.11. *Let K be a field with char$(K) \neq 2, 3$ and E an elliptic curve over K, defined by the Weierstrass equation $y^2 = x^3 + ax + b$.*

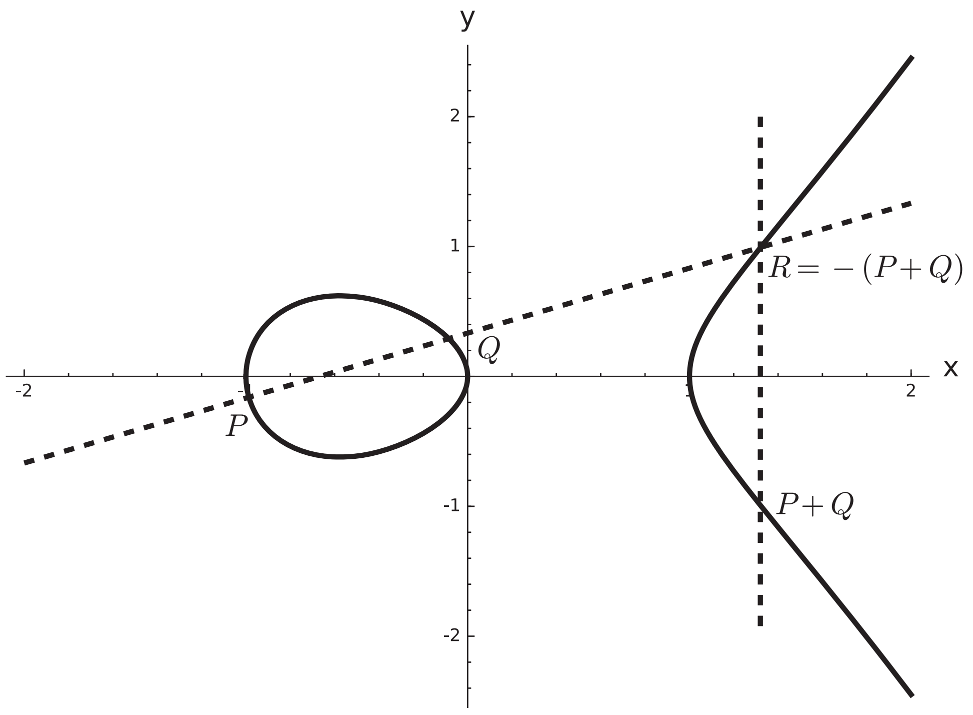


Figure 12.1. The elliptic curve $E : y^2 + y = x^3 - x$ over the real numbers. The line through P and Q intersects the curve in R and $P + Q + R = O$.

(1) *Let $P = (x_1, y_1)$ and $P \neq O$; then $-P = (x_1, -y_1)$.*

(2) *Let $P = (x_1, y_1)$, $Q = (x_2, y_2) \in E(K)$, $P \neq O$, $Q \neq O$ and $P \neq -Q$, i.e., $x_1 \neq x_2$ or $y_1 \neq -y_2$. Then*

$$P + Q = (x_3, y_3),\ x_3 = m^2 - x_1 - x_2,\ y_3 = m(x_1 - x_3) - y_1,$$

where m is the slope of the line through P and Q or the tangent line, if $P = Q$:

$$m = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q, \\ \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P = Q. \end{cases}$$

(3) *If $P = -Q$ then $P + Q = O$.*

(4) *If $P = O$ then $P + Q = Q$, and if $Q = O$ then $P + Q = P$.*

Example 12.12. Let $K = GF(19)$ and $E : y^2 = x^3 + 3x + 5$. The discriminant of E is $\Delta = -16(4 \cdot 3^3 + 27 \cdot 5^2) \equiv 12 \mod 19$, and so the Weierstrass curve is nonsingular. We verify that $P = (1, 3) \in E(K)$ and compute $2P$ using the above formulas. We obtain

$m = \frac{3+3}{2\cdot 3} = 1$, $x_3 = 1 - 1 - 1 \equiv 18 \mod 19$, $y_3 = (1-18) - 3 \equiv 18 \mod 19$, and so $2P = (18, 18)$. Furthermore, $-P = (1, -3) = (1, 16)$. ◊

Theorem 12.13. *$E(K)$ forms an abelian group with the above addition of points and the identity element $O = [0 : 1 : 0]$.*

Proof. We saw above that O is the identity element, that every point P has an inverse point $-P$ and that the addition is by construction commutative. It remains to be proven that the addition is associative, i.e., $(P+Q)+R = P+(Q+R)$ holds. This can be shown by tedious computations using the explicit formulas in Proposition 12.11, by geometric arguments (see [**Was08**]) or with more advanced results on algebraic curves (see [**Sil09**]). □

Example 12.14. Consider the elliptic curve $E : y^2 = x^3 + 3x + 5$ over $K = GF(19)$ (see Example 12.12). We let SageMath find all points on this curve.

```
sage: E=EllipticCurve(GF(19),[3,5])
sage: E.points()
[(0 : 1 : 0), (0 : 9 : 1), (0 : 10 : 1), (1 : 3 : 1), (1 : 16 : 1),
(2 :0 : 1), (4 : 9 : 1), (4 : 10 : 1), (6 : 7 : 1), (6 : 12 : 1),
(8 : 3 : 1), (8 : 16 : 1), (9 : 1 : 1), (9 : 18 : 1), (10 : 3 : 1),
(10 : 16 : 1), (11 : 1 : 1), (11 : 18 : 1), (14 : 6 : 1),
(14 : 13 : 1), (15 : 9 : 1), (15 : 10 : 1), (16 : 8 : 1),
(16 : 11 : 1), (18 : 1 : 1), (18 : 18 : 1)]
```

We see that $E(K)$ is an abelian group of order 26. By Theorem 4.29, $E(K)$ is isomorphic to $\mathbb{Z}_{26} \cong \mathbb{Z}_{13} \times \mathbb{Z}_2$. The points in $E(K)$ are depicted in Figure 12.2. Note that there is one extra point $O = [0 : 1 : 0]$ at infinity. All points must have order 1, 2, 13 or 26. Let $P = (1, 3) \in E(K)$. We use SageMath to compute $2P$ and $13P$.

```
sage: E=EllipticCurve(GF(19),[3,5]); P=E(1,3)
sage: 2*P
(18 : 18 : 1)
sage: 13*P
(2 : 0 : 1)
```

Since $2P \neq O$ and $13P \neq O$, the point P must have maximal order 26 and therefore generates $E(K)$. ◊

For cryptographic use, one chooses a finite field $K = GF(p)$ or $K = GF(2^m)$, an elliptic curve E over K and a *base point* $g \in E(K)$. The point g generates a cyclic subgroup $G = \langle g \rangle \subset E(K)$ of order $n = \text{ord}(g)$. n should be a large prime or at least contain a large prime factor. The cofactor is defined as $h = \frac{\text{ord}(E(K))}{n}$ and usually h is small or equal to 1.

Determining the order of a point g and the order of $E(K)$ is a non-trivial task, but there are efficient algorithms (see [**Was08**]). Hasse's Theorem provides the approximate number of points on an elliptic curve over a finite field:

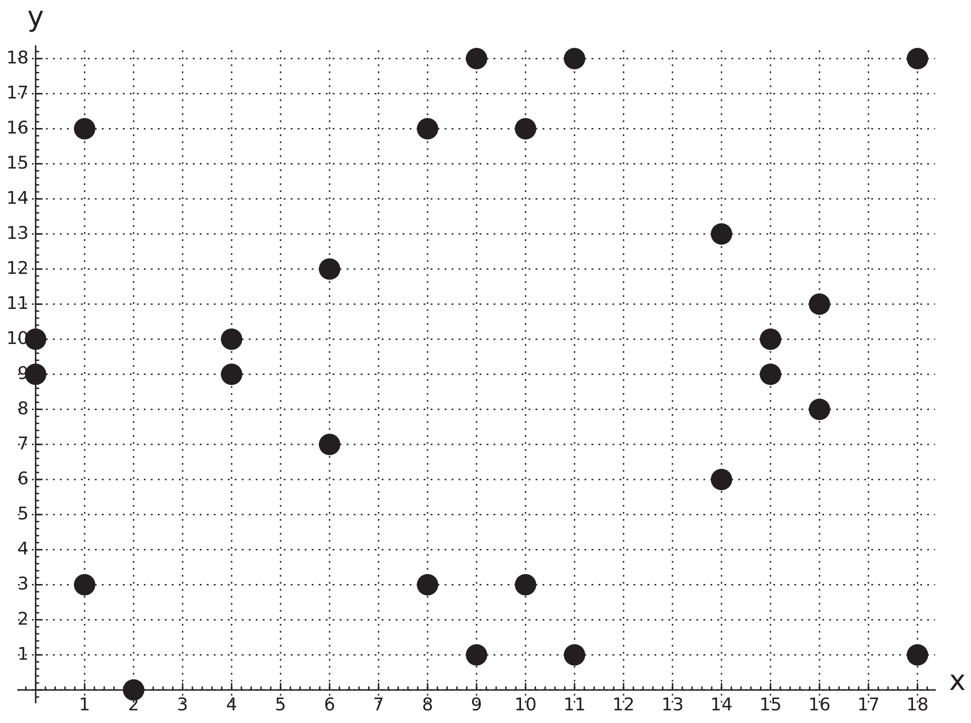


Figure 12.2. Points on the elliptic curve $y^2 = x^3 + 3x + 5$ over $GF(19)$.

Theorem 12.15. *Let E be an elliptic curve over a finite field K of order q. Then*

$$|\, q + 1 - \operatorname{ord}(E(K))\, | \le 2\sqrt{q}. \qquad \Diamond$$

Note that the obvious estimate based on the Weierstrass equation only gives $1 \le \operatorname{ord}(E(K)) \le 2q + 1$.

Example 12.16. Let E be any elliptic curve over $GF(19)$. Then $q + 1 = 20$ and $2\sqrt{q} \approx 8.7$. Hence $E(GF(19))$ must have between 12 and 28 points. In Example 12.14, we saw that the order of $E(GF(19))$ is 26.

Definition 12.17. Let E be an elliptic curve over a finite field K, $g \in E(K)$ a base point, $G = \langle g \rangle$, $n = \operatorname{ord}(G)$ and $A \in G$. Then the unique positive integer $a < n$ such that $a \cdot g = A$ is called the *discrete logarithm* $\log_g(A)$ of $A \in G$. $\Diamond$

Note that we use the term *discrete logarithm* although the group operation on $E(K)$ is written *additively*.

The security of elliptic curve cryptography relies on the hardness of the discrete logarithm (DL) problem (see Section 10.3) in the group $G \subset E(K)$. The elliptic curve and the parameters must be carefully chosen, since there are less secure curves where the computation of discrete logarithms can be reduced to an easier DL problem (see Section 12.3 below). The construction of *secure elliptic curves* and their *domain parameters* is beyond the scope of this book.

Elliptic curve cryptography is widely standardized by national and international organizations (e.g., ISO, ANSI, NIST, IEEE, IETF), and one of the proposed curves is usually chosen.

Example 12.18. We consider the domain parameters of the curve `brainpoolP256r1` (RFC 5639 [**LM10**]). The curve is defined by the Weierstrass equation $y^2 = x^3 + ax + b$ over a 256-bit field $K = GF(p)$. The base point $g = (x_g, y_g)$ generates the full group $G = E(K)$ and $n = \operatorname{ord}(g)$ is a 256-bit prime number.

```
p =  A9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
a =  7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9
b =  26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6
g =  (xg,yg)
xg=  8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE3262
yg=  547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997
n  = A9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7
h  = 1
```

These parameters can be verified using SageMath (see Exercise 4).

12.2. Elliptic Curve Diffie-Hellman

We know from Section 10.3 that the Diffie-Hellman key exchange is based on a cyclic group G. The common choice are subgroups of the multiplicative group $\mathbb{Z}_p^*$, but over the last decade, elliptic curves have become increasingly popular.

For a Diffie-Hellman key exchange with elliptic curves, the communication partners (say Alice and Bob) have to agree on a finite field K, an elliptic curve E over K and a base point g generating a group G of order n. Usually, they would choose a standard curve and its domain parameters as in Example 12.18.

Alice and Bob generate uniform secret keys a and $b \in \mathbb{Z}_n$, respectively. Alice computes $A = a \cdot g$ and Bob $B = b \cdot g$; then they exchange the public values A and B. Subsequently, Alice and Bob can both compute the shared secret key $k = (ab) \cdot g = a \cdot B = b \cdot A \in G$. The x-coordinate of a key $k = (x, y)$ can be used as a shared secret. Note that x determines a quadratic equation for y and there are at most two solutions for y. For a uniform output, the x-coordinate is taken as input of a key derivation or hash function.

An eavesdropper, who only knows A and/or B as well as the elliptic curve and its domain parameters, should not be able to derive a, b or k if the computational Diffie-Hellman (CDH) problem (see Section 10.3) is hard in G.

Example 12.19. Alice and Bob agree on the elliptic curve $y^2 = x^3+3x+5$ over $GF(19)$, and the base point is $g = 2 \cdot (1,3) = (18,18)$. The point g has order 13. Alice chooses the secret key $a = 2$ and computes $A = a \cdot g = 2 \cdot (18,18) = (11,18)$. Bob chooses the secret key $b = 4$ and computes $B = b \cdot g = 4 \cdot (18,18) = (8,3)$. They exchange A and B.

Alice obtains the shared secret key by computing $k = a \cdot B = 2 \cdot (8,3) = (9,1)$ and Bob computes $k = b \cdot A = 4 \cdot (11,18) = (9,1)$.

Remark 12.20. Elliptic curve Diffie-Hellman can also be used as a *key encapsulation mechanism* (KEM) as explained in Section 10.6. The encapsulated key is $H(k)$, where $H : G \to \{0,1\}^l$ is a key derivation (or hash) function on G (or on K if the x-coordinate is used). The KEM also gives a hybrid elliptic curve encryption scheme called ECIES (see Section 10.7).

12.3. Efficiency and Security of Elliptic Curve Cryptography

Diffie-Hellman and other algorithms which use elliptic curves require the multiplication of points. Analogous to the square-and-multiply algorithm for a fast exponentiation (see Section 3.3), multiplying a point by a factor can be done recursively by doubling and adding points (*double-and-add algorithm*). Proposition 12.11 shows that the doubling or addition of points requires a few additions and multiplications as well as division over the base field K. For $K = GF(p)$, the running time of a single addition or doubling of points is $O(\text{size}(p)^2)$. Using the double-and-add algorithm, the complexity of multiplying a point by a factor of a is $O(\text{size}(a)\ \text{size}(p)^2)$.

Suppose that $G = E(GF(p))$ and $n = \text{ord}(G)$. By Hasse's Theorem 12.15, the order of G is about p and hence the computational complexity of multiplications in G is $O(\text{size}(p)^3)$. This is the same cubic complexity as in other public-key algorithms, for example RSA and Diffie-Hellman over $\mathbb{Z}_p^*$. However, p can be relatively small in ECC, for example 256 bits long, and achieve a similar level of security as a much larger multiplicative group $\mathbb{Z}_p^*$, say with $\text{size}(p) = 3072$.

The *security* of elliptic curve schemes relies on the hardness of the discrete-logarithm (DL) problem. A major difference to the multiplicative group is that the known sub-exponential algorithms (in particular Index-Calculus) cannot be applied to elliptic curves. If $n = \text{ord}(G)$ is a prime number, then the best known algorithms for computing discrete logarithms on elliptic curves are Babystep-Giantstep and Pollard's ρ-method for logarithms (see Section 10.5). Both algorithms have exponential complexity $O(\sqrt{n})$. However, the elliptic curve needs to be carefully chosen in order to prevent certain types of attacks.

Remark 12.21. Let $G = \langle g \rangle \subset E(GF(p))$ and $n = \text{ord}(g)$ be as above and suppose we have a group homomorphism $f : G \to G'$. With $m = \text{ord}(f(g))$ we have

$m \mid n$. Let $a \in \{0, 1, \dots, n-1\}$ and $A = a \cdot g$. We obtain $f(A) = a \cdot f(g)$, and thus $\log_g(A) \equiv \log_{f(g)}(f(A)) \mod m$. Now the approach is to compute the discrete logarithm in the group G' instead of G, which might be easier. In fact, using the Weil or Tate-Lichtenbaum pairing, G can be efficiently embedded into the *multiplicative group* of an extension field of $GF(p)$. We refer to [**Was08**] for pairings and the *MOV attack*, named after Menezes, Okamoto and Vanstone [**MOV93**], against the DL problem for elliptic curves. ◊

Below, we list some of the requirements (see RFC 5639 [**LM10**]):

- Using the Weil or Tate-Lichtenbaum pairing mentioned above, one can reduce the DL problem in G to the DL problem in the multiplicative group of the field $GF(p^l)$, where $n \mid p^l - 1$, or equivalently, $p^l \equiv 1 \mod n$. However, this can only be exploited if l is small enough so the discrete logarithm in $GF(p^l)^*$ can be efficiently computed. In order to prevent this attack, one requires that the multiplicative order of $p \mod n$ is large and close to the maximum possible value $\varphi(n)$.
- Anomalous curves have exactly p points over $GF(p)$. Such curves must be avoided, since the DL problem can be reduced to an additive group, where computation of the discrete logarithm is a simple division and thus runs in polynomial time.
- The group order $n = \operatorname{ord}(G)$ should be a prime number, so that the DL problem cannot be reduced to subgroups of G (Pohlig-Hellman algorithm).
- The generation of the elliptic curve and its domain parameters should be pseudorandom.

Table 12.1 (see [**Bar16**]) shows comparable strengths for different algorithms and key lengths (in bits). The table compares symmetric encryption schemes (key size), RSA (size of the modulus N), Diffie-Hellman using a subgroup of order q in $\mathbb{Z}_p^*$ (size of p, size of q) and Diffie-Hellman with elliptic curves (size of n, where n is the order of the cyclic group G).

Table 12.1. Comparable algorithm strengths for different key lengths.

Symmetric Encryption	RSA	DH	ECDH
80	1024	1024, 160	160 – 223
112	2048	2048, 224	224 – 255
128	3072	3072, 256	256 – 383
192	7680	7680, 384	384 – 511
256	15360	15360, 512	512+

12.4. Elliptic Curve Factoring Method

Surprisingly, elliptic curves cannot only be used to secure cryptographic operations, but also to attack RSA by *factoring large integers*.

Lenstra's *elliptic curve factoring method* (ECM) is similar to *Pollard's* $p-1$ *method* (see Section 9.7). Suppose that $N = pq$ is given and the factors p, q are unknown. If all prime factors of $p-1$ are small, then one has a significant chance of finding a multiple k of $p-1$, for example by setting $k = B!$ for some bound B. Euler's Theorem implies $a^{p-1} \equiv 1 \mod p$ and hence $a^k \equiv 1 \mod p$ for any base a that is not a multiple of p. Hence $\gcd(a^k-1, N)$ is either p (success) or N (failure). Pollard's method is successful if the image of a^k in

$$\mathbb{Z}_N^* \cong \mathbb{Z}_p^* \times \mathbb{Z}_q^*$$

is congruent to 1 in exactly one of the components. This attack can be prevented by choosing primes p such that $p-1$ contains a large prime factor (strong primes). Randomly chosen primes p usually satisfy this condition.

ECM replaces the multiplicative group $\mathbb{Z}_N^*$ in Pollard's method by the group of points on an elliptic curve E over $\mathbb{Z}_N$. Although $\mathbb{Z}_N$ is only a ring and not a field, we have the decompositions

$$\mathbb{Z}_N \cong \mathbb{Z}_p \times \mathbb{Z}_q \text{ and } E(\mathbb{Z}_N) \cong E(GF(p)) \times E(GF(q)).$$

The formulas for addition and doubling of points in Proposition 12.11 (2) also hold over $\mathbb{Z}_N$ if the point is not equal to O modulo p or q. We choose an elliptic curve E and a point $P \in E(\mathbb{Z}_N)$. Then compute kP, for example $k = B!$, and check whether the result exists as an affine point modulo N. This fails if and only if a denominator of a slope m in the computation of kP is not invertible modulo N. In this case, the greatest common divisor (gcd) of the denominator of m and N is not equal to 1, and it is very likely that the gcd is either p or q, and not N.

One may proceed as follows to choose an elliptic curve E and a point P over $\mathbb{Z}_N$: choose random integers a, u and v between 0 and $N-1$. Let

$$\begin{aligned} P &= (u, v) \mod N, \\ b &= v^2 - u^3 - au \mod N, \\ E : y^2 &= x^3 + ax + b. \end{aligned}$$

By construction we have $P \in E(\mathbb{Z}_N)$. Then define $k = B!$ for some bound B and compute kP. If kP does not exist modulo N (as an affine point, see Remark 12.22 below), then one has found a non-trivial factor of N, i.e., either p, q or N. If kP exists, or if the factor is N, then choose a new random curve E and point P or increase B and start over.

Remark 12.22. We say that the point *kP does not exist modulo N* if it cannot be computed modulo N (because of a non-invertible denominator). However, the corresponding *projective point* exists and the reduction modulo p or modulo q is equal to the point O at infinity. The aim of ECM is to produce such a *non-affine* point, since it yields a non-trivial factor of N. This is analogous to Pollard's $p-1$ method, where one is looking for a power a^k that is congruent to 1 modulo p or modulo q.

Example 12.23. Let $N = 1211809$. We choose $a = 10$, $u = 5$ and $v = 7$ so that $b = -126$. The corresponding elliptic curve is

$$E : y^2 = x^3 + 10x - 126$$

and $P = (5, 7) \in E(\mathbb{Z}_N)$. Now we compute multiples of P. We choose the bound $B = 10$, set $k = B!$ and hope that the order of P in $E(GF(p))$ or $E(GF(q))$ contains only prime factors less than or equal to B. We compute

$$2P,\ (3!)P,\ (4!)P,\ \dots,\ (10!)P$$

and stop if any of these multiples does not exist as an affine point modulo N. The reader can reproduce the following computations using SageMath.

We omit some intermediate steps and obtain

$$(6!)P = 720 \cdot P = (222064, 820051).$$

We denote this point by $Q = (x_1, y_1)$. The next step is computing $(7!)P = 7Q$. Using the double-and-add algorithm and the formulas in Proposition 12.11, we compute $2Q = (884483, 792125)$, $3Q = 2Q + Q = (179208, 246408)$ and $6Q = 2 \cdot (3Q) = (1011121, 433793) = (x_2, y_2)$. Finally, $7Q = 6Q + Q$, but now the slope

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

does not exist modulo N since the denominator

$$x_2 - x_1 = 1011121 - 222064 = 789057$$

has a common factor with N. In fact, $\gcd(x_2 - x_1, N) = 1201 = p$, and thus we have found a factor of N. The second factor is $q = 1009$.

We see that the elliptic curve factoring method is successful for N and the chosen curve E, point P and multiple $k = 7!$. After having found $p = 1201$, it is not difficult to understand why the method is successful: the group $E(GF(1201))$ has order $1176 = 2^3 \cdot 2 \cdot 7^2$, which contains only prime factors less than or equal to 7 so that $(7!)P = O \mod p$.

On the other hand, the order of $E(GF(q))$ is $1041 = 3 \cdot 347$, and hence $(7!)P \neq O \mod q$. So we obtain only the factor p and not the product $pq = N$. ◊

ECM is very successful in factoring integers N with less than around 80 decimal digits. For larger values of N, the sieve methods (see Section 9.7) are more efficient.

12.5. Summary

- An elliptic curve is a nonsingular projective curve defined by a cubic Weierstrass equation in two variables. The points of an elliptic curve over a field form an abelian group.
- The group of points on an elliptic curve over a finite field can be used in public-key schemes. Common applications of elliptic curves are the Diffie-Hellman key exchange, key encapsulation, hybrid encryption and digital signatures.
- The security of elliptic curve cryptography relies on the assumed hardness of the discrete logarithm problem. The elliptic curve and the domain parameters need to be carefully chosen.
- Elliptic curve schemes have shorter keys than classical public-key algorithms with comparable strength.
- Elliptic curves can also be used to factorize large integers.

Exercises

1. Let E be the elliptic curve defined by $y^2 = x^3 + 3x + 5$.
   (a) Assume that E is defined over $K = GF(19)$. Compute the discriminant Δ. Show that $P = (4, 9) \in E(GF(19))$ and compute $2P$.
   (b) Now assume that E is defined over $K = \mathbb{Q}$. Compute the discriminant Δ. Show that $P = (4, 9) \in E(\mathbb{Q})$ and compute $2P$.
2. Suppose E is an elliptic curve over $K = GF(23)$.
   (a) Give the minimum and the maximum number of points on $E(K)$.
   (b) How many point additions or doublings are at most needed to compute nP for any $n \in \mathbb{Z}$ and $P \in E(K)$?
3. Consider the Weierstrass curve $y^2 = x^3$ and let $E_{ns}(K)$ be the set of nonsingular points on the curve (see Example 12.8). Prove that the map
$$f : E_{ns}(K) \to (K, +),\ (x, y) \mapsto \frac{x}{y},\ O \mapsto 0$$
is bijective. Add-on: Use the addition formulas in Proposition 12.11 to show that f is a homomorphism, i.e., $f(P_1 + P_2) = f(P_1) + f(P_2)$ holds. Therefore, f is an isomorphism.
Tip: Let $t = \frac{x}{y}$. Verify that $x = \frac{1}{t^2}$ and $y = \frac{1}{t^3}$.
4. Show that the parameters in Example 12.18 are valid, i.e., that p and n are prime, $\Delta \neq 0$ and $\text{ord}(g) = n$. Furthermore, show that $\text{ord}(p \mod n)$ is large.
Hint: Use SageMath. Hexadecimal numbers can be defined using the prefix `0x`. Check the primality using `is_prime()`. Define the elliptic curve `E=EllipticCurve(GF(p),[a,b])` and the point `g=E(xg,yg)`. The order of $E(K)$ can be obtained by `E.order()`.

5. Use the parameters in Example 12.18 for an elliptic curve Diffie-Hellman key exchange. Assume that Alice and Bob choose the following secret parameters:
```
a=81DB1EE100150FF2EA338D708271BE38300CB54241D79950F77B063039804F1D
b=55E40BC41E37E3E2AD25C3C6654511FFA8474A91A0032087593852D3E7D76BD3
```
Compute A, B, bA, aB and k.
6. Factorize $N = 6227327$ using the elliptic curve factoring method and SageMath.
   (a) Choose $a = 4$, $u = 6$ and $v = 2$. Give the associated elliptic curve over $\mathbb{Z}_N$ and the point $P \in E(\mathbb{Z}_N)$.
   Tip: `E=EllipticCurve(IntegerModRing(N),[a,b])` and `P=E(u,v)`.
   (b) Let $B = 13!$. Show that $(12!)P$ exists (as an affine point), but not $(13)!\,P$.
   (c) Find the critical denominator and compute the gcd with N.
   (d) Give the factors p and q of N and explain why the method is successful with the chosen parameters.
7. The *Elliptic Curve Digital Signature Algorithm* (ECDSA) is the elliptic curve analogue of DSA (see Exercise 11.10) and also standardized in [**FIP13**]. The scheme uses an elliptic curve E over a finite field K and a base point g of prime order n. Choose a uniform secret key $a \in \mathbb{Z}_n$ and compute the point $A = a\,g \in E(K)$. The domain parameters of the elliptic curve and A form the public key. Similarly to ElGamal and DSA, the signature is randomized and also requires a hash function H. In order to sign a message m, a secret uniform integer k with $1 \leq k \leq n-1$ is chosen and $k\,g$ is computed. Let r be the x-coordinate modulo n of the point $k\,g$. If $r = 0$ then choose a new value k. Otherwise, let
$$s = k^{-1}(H(m) + ar) \bmod n.$$
If $s = 0$ then start again with a new value k. Otherwise, the pair (r, s) is the signature of m.

   To verify the signature (r, s) of a message m, one checks that $1 \leq r \leq n-1$ and $1 \leq s \leq n-1$. Then compute $s^{-1} \bmod n$, $s^{-1}H(m) \bmod n$, $s^{-1}r \bmod n$ and the point
$$R = s^{-1}H(m)g + s^{-1}rA \in E(K).$$
Since $\operatorname{ord}(g) = n$, the result does not depend on the representatives of $s^{-1}H(m)$ and $s^{-1}r$ modulo n. If $R = O$ then the signature is invalid. Otherwise, reduce the x-coordinate of R modulo n. The signature is valid if the result is r.
   (a) Prove that the verification is correct.
   (b) Use the elliptic curve and the parameters of Example 12.19; the base point is $g = (18, 18)$ and $n = \operatorname{ord}(g) = 13$. Alice's secret key is $a = 2$. She wants to sign a message m with $H(m) \equiv 11 \bmod n$ and chooses $k = 3$. Compute the signature (r, s) and verify the signature using her public key.
   (c) Show that k must remain secret.

Chapter 13

Quantum Computing

This chapter provides an introduction to *quantum computing*. We focus on quantum computation and cryptographic applications, not on physical realizations of quantum computers, and we do not require any previous knowledge in quantum mechanics. Quantum computers have amazing features: they can process a very large number of input values simultaneously and solve certain hard problems. At the time of this writing, quantum computers are not yet large and stable enough to break any real-world cryptosystems, but this might change within a decade. Today's quantum systems are noisy and quantum error correction is an important topic. There is now a broad range of research into post-quantum cryptography, and new schemes should be secure in the presence of quantum computers. Chapters 14 and 15 give an introduction to several post-quantum schemes.

The basic information unit of quantum computing is a quantum bit (qubit), which can assume not only two, but infinitely many states. However, only one classical bit can be extracted (or measured) from a qubit and the output is probabilistic. In Section 13.1, qubits and operations on single qubits are introduced. Systems with multiple qubits are dealt with in Section 13.2. Analogous to classical computing, quantum algorithms are realized with quantum gates. Since the output is probabilistic, quantum algorithms need to be carefully designed in order to obtain useful results. The Deutsch-Josza algorithm is explained in Section 13.3 and demonstrates the possibilities of quantum computing.

A major algorithm is the Quantum Fourier Transform (Section 13.4), which can efficiently find periods in a large (exponential) set of numbers. In Section 13.5, we show that Shor's algorithm can efficiently factorize integers and break the RSA cryptosystem.

Quantum bits can also be used for a secure key distribution, and we explain the BB84 quantum cryptographic protocol in Section 13.6.

Two recommended textbooks for further reading on quantum computing are [**NC00**] and [**RP11**].

13.1. Quantum Bits

A classical computer processes the input data sequentially in order to solve a problem. Turing machines or logical circuits can be used to model classical computation. A Boolean circuit is made up of elementary gates and performs a computational task, where input bits are transformed into output bits. In each step, the system has a fixed state and, depending on the complexity of an algorithm and the size of the input data, the computation can be inefficient or infeasible: suppose that f is a vectorial Boolean function and an algorithm iterates over all $x \in \{0,1\}^n$ until $f(x)$ is equal to a fixed value y. The worst-case running time is exponential in n and the algorithm is inefficient on conventional computers, even if the individual computations of $f(x)$ run very fast.

Now, quantum computers can compute *all* 2^n *values* $f(x)$ *simultaneously*. This sounds fantastic, but there is a catch: the computation requires a system of n qubits, the internal state is inaccessible and only a single output value can be extracted from a quantum system. Nevertheless, we will see that quantum algorithms can efficiently solve many hard problems.

Quantum computing uses *quantum circuits* instead of Boolean circuits, and quantum algorithms take *qubits* (quantum bits) instead of classical bits as input. A qubit is a two-level quantum-mechanical system and can have different physical implementations, for example the polarization of a photon, the spin of an electron or the state of an ion. The physical realization of quantum systems is an important field of research and development, but it is beyond the scope of this book.

A qubit can assume infinitely many states between 0 or 1; the state is represented by a normalized vector in the vector space $\mathbb{C}^2$. We fix an orthonormal basis of $\mathbb{C}^2$, for example the standard basis $e_1 = (1,0)$ and $e_2 = (0,1)$, and denote the basis states by $|0\rangle$ and $|1\rangle$. The *Dirac* or *ket* notation of a vector ψ in the state space is $|\psi\rangle$. The state of a qubit $|\psi\rangle$ is a *superposition* (linear combination) of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle\,, \text{ where } a,\ b \in \mathbb{C} \text{ and } |a|^2 + |b|^2 = 1.$$

The coefficients of $|0\rangle$ and $|1\rangle$ can be interpreted as probabilities, and a qubit as a random variable. A measurement (observation) changes the state of a qubit and yields a classical bit, i.e., either 0 or 1. The state of a qubit determines the probability of the result of a measurement: the probability of 0 is $|a|^2$ and the probability of 1 is $|b|^2$. The original state of a qubit, and hence the amplitudes a and b, are lost after the measurement. Thus the quantum information (a and b) is hidden and cannot be directly extracted. However, the quantum information can be used in quantum circuits that transform the state of qubits.

Example 13.1. The measurement of a qubit having the state $|0\rangle = 1 \cdot |0\rangle + 0 \cdot |1\rangle$ always gives 0, and the measurement of the state $|1\rangle$ always gives 1. On the other hand, if a qubit has the state

$$\frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle,$$

then the probability of both 0 and 1 is $\frac{1}{2}$, and a measurement outputs a uniform random bit. This above state is quite useful and we denote it by $|+\rangle$. ◊

A geometric representation of the state of a single qubit is given by the *Bloch sphere* (see Figure 13.1). The points on a two-dimensional unit sphere in the three-dimensional space $\mathbb{R}^3$ are given by two angles, the polar angle $\theta \in [0, \pi]$ and the azimuth $\varphi \in]-\pi, \pi]$. In geography, the polar angle and the azimuth are called *colatitude* and *longitude*, respectively.

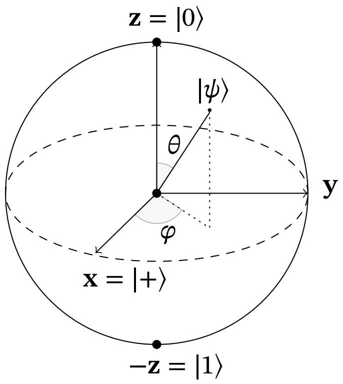


Figure 13.1. Bloch sphere.

How can we represent a state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ by a point on the Bloch sphere? We use the fact that the state of a qubit does not depend on a global phase γ:

$$|\psi\rangle \sim e^{i\gamma}\,|\psi\rangle.$$

We may thus assume that the *phase* of the first coefficient a is zero, so that a is a non-negative real number. We express the second coefficient $b \in \mathbb{C}$ in polar form:

$$b = re^{i\varphi},$$

where $r \geq 0$ and $\varphi \in]-\pi, \pi]$. The condition $|a|^2 + |b|^2 = 1$ implies $a^2 + r^2 = 1$, since $|e^{i\varphi}| = 1$. The non-negative parameters a and r thus lie on a unit circle in the first quadrant and we can write

$$a = \cos\left(\frac{\theta}{2}\right) \text{ and } r = \sin\left(\frac{\theta}{2}\right),$$

where $\theta \in [0, \pi]$. We can therefore rewrite $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ as

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle,$$

where $\theta \in [0, \pi]$ and $\varphi \in\,]-\pi, \pi]$.

Example 13.2. Here are the representations of several states:

$$\begin{aligned}
|0\rangle &= \cos(0)\,|0\rangle + e^{i\cdot 0}\sin(0)\,|1\rangle & \theta &= 0 & \varphi &= 0,\\
|1\rangle &= \cos\left(\frac{\pi}{2}\right)|0\rangle + e^{i\cdot 0}\sin\left(\frac{\pi}{2}\right)|1\rangle & \theta &= \pi & \varphi &= 0,\\
|+\rangle &= \cos\left(\frac{\pi}{4}\right)|0\rangle + e^{i\cdot 0}\sin\left(\frac{\pi}{4}\right)|1\rangle & \theta &= \frac{\pi}{2} & \varphi &= 0,\\
|-\rangle &= \cos\left(\frac{\pi}{4}\right)|0\rangle + e^{i\cdot \pi}\sin\left(\frac{\pi}{4}\right)|1\rangle & \theta &= \frac{\pi}{2} & \varphi &= \pi.
\end{aligned}$$

Note that a global phase does not change the state, e.g., $i\,|+\rangle \sim |+\rangle$, but the relative phase φ is important! For example, $|+\rangle$ and $|-\rangle$ are different states. ◊

Next, we study operations on a single qubit by a *quantum gate*.

Definition 13.3. A *quantum gate U_f* with a single input and output qubit is described by a *unitary* 2×2 matrix

$$U = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

A state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ is transformed into

$$U_f\,|\psi\rangle = U_f(a\,|0\rangle + b\,|1\rangle) = (c_{11}a + c_{12}b)\,|0\rangle + (c_{21}a + c_{22}b)\,|1\rangle.$$ ◊

Surprisingly, there is more than one nontrivial operation on a single qubit.

Definition 13.4. (1) The quantum analogue of the classical `NOT` gate transforms the state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ into $|\overline{\psi}\rangle = b\,|0\rangle + a\,|1\rangle$. The swapping of coefficients is given by the *Pauli-X* matrix

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

(2) The *Hadamard* gate is described by the unitary matrix

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$ ◊

Figure 13.2 depicts the Pauli-X and Hadamard gates.

Since $H \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, the state $|0\rangle$ is transformed into $|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$. This is very useful in order to produce a balanced superposition of the basis states. Loosely speaking, a 0-bit is turned into a qubit that is simultaneously 0 and 1. Measuring $H\,|0\rangle$ gives a uniform random bit (see Figure 13.3).

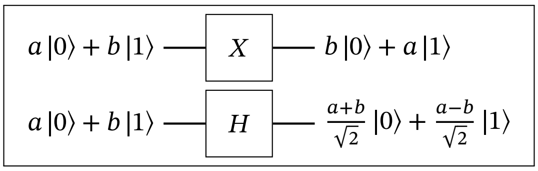

$a|0\rangle + b|1\rangle$
X
$b|0\rangle + a|1\rangle$
$a|0\rangle + b|1\rangle$
H
$\frac{a+b}{\sqrt{2}}|0\rangle + \frac{a-b}{\sqrt{2}}|1\rangle$

Figure 13.2. Pauli-X and Hadamard gates.

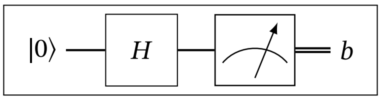


Figure 13.3. Measuring $H|0\rangle$ gives a uniform random bit b.

Definition 13.5. Other commonly used single qubit gates are:

(1) *Pauli-Y gate* with matrix $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$.

(2) *Pauli-Z gate* with matrix $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.

(3) *Phase gate* with matrix $S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$.

(4) $\frac{\pi}{8}$ *gate* with matrix $T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$. ◊

Since $Y = -iZX$, Pauli-Y is a composition of Pauli-X and Pauli-Z. The gates Pauli-Z, Phase and $\frac{\pi}{8}$ change the relative phase by π, $\frac{\pi}{2}$ and $\frac{\pi}{4}$, respectively. On the Bloch sphere, these gates give rotations of π, $\frac{\pi}{2}$ and $\frac{\pi}{4}$ around the z-axis. For example, one has

$$Z|\psi\rangle = Z\left(\cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle\right) = \cos\left(\frac{\theta}{2}\right)|0\rangle - e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle .$$

This state can be written as

$$Z|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i(\varphi+\pi)}\sin\left(\frac{\theta}{2}\right)|1\rangle .$$

Hence the Z gate adds π to the azimuth φ on the Bloch sphere.

The reason for the historical name $\frac{\pi}{8}$ *gate* is the fact that T is (up to an unimportant global phase) equal to a matrix with $\pm\frac{\pi}{8}$ on its diagonals:

$$T = e^{i\pi/8}\begin{pmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{pmatrix}.$$

13.2. Multiple Qubit Systems

More interesting quantum operations require systems of *multiple qubits*. A composite system of n qubits can represent 2^n states simultaneously. The basis states are

$$|x_1 x_2 \dots x_n\rangle ,$$

where $x_i \in \{0,1\}$. The general state of an n-qubit system is a superposition of the 2^n basis states. Such a system is not the same as n individual qubits!

Remark 13.6. The state of system of n qubits is represented by the n-fold *tensor product* of $\mathbb{C}^2$:

$$\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2 = (\mathbb{C}^2)^{\otimes n}.$$

The general construction of tensor products is beyond our scope, but in this case $(\mathbb{C}^2)^{\otimes n}$ is a $\mathbb{C}$-vector space of dimension 2^n. The elements in $(\mathbb{C}^2)^{\otimes n}$ are linear combinations of vectors $v_1 \otimes v_2 \otimes \cdots \otimes v_n = |v_1, v_2, \dots, v_n\rangle$, where $v_i \in \mathbb{C}^2$. The tensor product is linear in each component. The standard basis states are given by

$$x_1 \otimes x_2 \otimes \cdots \otimes x_n = |x_1 x_2 \dots x_n\rangle ,$$

where $x_i = |0\rangle$ or $|1\rangle$. A general vector in $(\mathbb{C}^2)^{\otimes n}$ can be written as

$$v = \sum_{x \in \{0,1\}^n} a_x |x_1 x_2 \dots x_n\rangle .$$

The norm of v is the non-negative real number

$$||v|| = \sum_{x \in \{0,1\}^n} |a_x|^2.$$

The normalization condition for multiple qubit systems requires that $||v|| = 1$. Unitary operators $U_{f_1}, \dots, U_{f_n}$ on $\mathbb{C}^2$ induce the unitary operator

$$U_f = U_{f_1} \otimes \cdots \otimes U_{f_n}$$

on $(\mathbb{C}^2)^{\otimes n}$, but there are additional operators that are not of this type.

As with single qubits, a multiple qubit state does not depend on a global phase and

$$|\psi\rangle \sim e^{i\gamma} |\psi\rangle .$$

A two qubit system is represented by a state in $\mathbb{C}^2 \otimes \mathbb{C}^2$. The four basis states are $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$, and a general state is a superposition of the four basis states:

$$|\psi\rangle = a_{00} |00\rangle + a_{01} |01\rangle + a_{10} |10\rangle + a_{11} |11\rangle .$$

The normalization condition requires $|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$. Measuring a two qubit system gives 00 with probability $|a_{00}|^2$, 01 with probability $|a_{01}|^2$, 10 with probability $|a_{10}|^2$ and 11 with probability $|a_{11}|^2$.

Example 13.7. (1) Consider a two qubit system in the state

$$|\psi\rangle = \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)$$
$$= \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle.$$

All four combinations have the same probability $(\frac{1}{2})^2 = \frac{1}{4}$.

(2) One of the *Bell states* is

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle.$$

After a measurement, the state is either 00 or 11. The combinations 01 and 10 cannot occur; a measurement of the two qubits must give the same result. The qubits are *entangled*. In quantum mechanics, this is called the EPR (Einstein, Podolsky, Rosen) paradox, where two particles are mysteriously correlated. The other three Bell states are given in Exercise 2. ♢

One can show that the Bell states are not the product of any two single qubit states (see Exercise 1). In fact, two entangled qubits behave differently from two single qubits.

The basis states of a system of n qubits are $|x_1 x_2 \dots x_n\rangle$ where $x_i \in \{0,1\}$. A general state is a superposition

$$|\psi\rangle = \sum_{x\in\{0,1\}^n} a_x |x\rangle \text{ with } \sum_{x\in\{0,1\}^n} |a_x|^2 = 1.$$

A basis state $|x_1 x_2 \dots x_n\rangle$ can also be written as $|x\rangle$, where

$$x = x_1 2^{n-1} + x_2 2^{n-2} + \cdots + x_n \in \{0,\ 1,\ \dots,\ 2^n - 1\}.$$

Then, a general state of an n-qubit system is

$$|\psi\rangle = \sum_{x=0}^{2^n-1} a_x |x\rangle \text{ with } \sum_{x=0}^{2^n-1} |a_x|^2 = 1.$$

An obvious generalization of the Bloch sphere for multiple qubits is not known. Note that the full state of a system of n qubits involves 2^n complex amplitudes (and $2^n - 1$ degrees of freedom, since the amplitude vector is normalized and multiplication by a global phase is unimportant). This is a huge number, say for $n > 100$. We emphasize that a measurement outputs only *one binary word* x of length n and the probability of x being measured is $|a_x|^2$.

13.3. Quantum Algorithms

Classical Boolean circuits transform input bits into output bits. Not surprisingly, *quantum circuits* process qubits. The basic building blocks of quantum circuits are quantum logic gates, which implement unitary (and therefore reversible) transformations on qubits. This is a major difference to classical circuits, which are often not reversible, for example the elementary AND or OR gates. However, this does not pose a serious restriction, since there are invertible analogues of the classical gates. It turns out that every classical circuit has a quantum analogue, giving the same output on the basis states, but also processing superpositions of the basis states.

Definition 13.8. The *controlled-NOT* operation *CNOT* on two input qubits with basis states $|x\rangle$ and $|y\rangle$ is given by

$$CNOT\,|x, y\rangle = |x, x \oplus y\rangle$$

(see Figure 13.4). This transformation acts on the basis states as follows: the first bit (the *control bit*) is unchanged and the second bit (the *target bit*) is flipped if the control bit is 1. The states $|00\rangle$ and $|01\rangle$ are thus unchanged, $|10\rangle$ is mapped to $|11\rangle$ and $|11\rangle$ to $|10\rangle$. The CNOT gate is represented by the unitary matrix

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

since the first two basis states remain unchanged, while the third and the fourth basis state are swapped. ◊

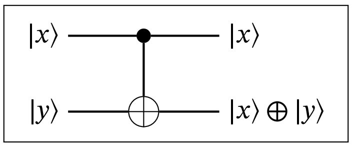


Figure 13.4. Controlled NOT gate.

CNOT can produce entangled states and this two-bit gate is not the tensor product of two single-qubit gates. For example, two qubits are transformed into the Bell state:

$$CNOT\left(\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)\right) = \frac{1}{\sqrt{2}}\,(|00\rangle + |11\rangle)\,.$$

CNOT is an example of a two qubit *controlled* gate (see Figure 13.5): if the first (control) qubit is $|1\rangle$, then a single qubit operation Q is performed on the second (target) qubit. If the control qubit is $|0\rangle$, then the target qubit is unchanged (see Figure 13.5). The controlled Q gate is represented by a 4×4 matrix of four 2×2 blocks, where I_2 is the 2×2 identity matrix:

$$\begin{pmatrix} I_2 & 0 \\ 0 & Q \end{pmatrix}.$$

CNOT is a controlled Q gate with $Q = X$ (Pauli-X). The CNOT gate is especially important, since single qubit and CNOT gates are *universal*.

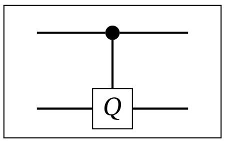


Figure 13.5. Controlled Q gate.

Theorem 13.9. *Single qubit gates and the CNOT gate are sufficient to implement an arbitrary unitary operation on n qubits.*

Proof. We refer to [**NC00**] for a proof. Firstly, one shows that an arbitrary unitary matrix can be decomposed into a product of *two-level* unitary matrices, where at most two coordinates are changed. Secondly, circuits from two-level unitary matrices are built from single qubit and CNOT gates. ☐

Remark 13.10. At the time of writing, IBM offers quantum computers and simulators for public use (`https://quantum-computing.ibm.com`). You can create and run your own algorithms on quantum devices using a graphical composer or a Quantum Assembly Language Code (QASM) editor. Circuits are built from a set of elementary gates (X, Y, Z, H, S, T, CNOT, ...), and combinations of them can express more complicated unitary transformations. Also look at the open source framework Qiskit (`https://qiskit.org`) for working with noisy quantum computers.

Many quantum algorithms take as input a superposition of the basis states. The single qubit Hadamard gate H (see Section 13.1) transforms the basis state $|0\rangle$ into the superposition

$$\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right).$$

The *Walsh-Hadamard transformation* generalizes this to a system of n qubits. It can be implemented by n parallel Hadamard gates.

Definition 13.11. The Walsh-Hadamard transformation W acts on a system of n qubits and is defined by

$$W = H \otimes H \otimes \cdots \otimes H = H^{\otimes n}. \qquad \diamondsuit$$

The Walsh-Hadamard transformation W transforms the zero state into a balanced superposition of all 2^n basis states (see Figure 13.6):

$$\begin{aligned} H^{\otimes n}|00\ldots0\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \otimes \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \otimes \cdots \otimes \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \\ &= \frac{1}{\sqrt{2^n}}\left(|00\ldots0\rangle + |00\ldots1\rangle + \cdots + |11\ldots1\rangle\right). \end{aligned}$$

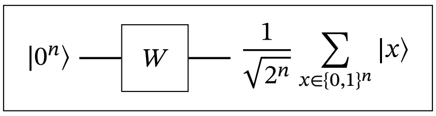


Figure 13.6. Walsh-Hadarmard transformation.

The second equation follows from the fact that the tensor product is linear in each component.

Quantum algorithms can use this superposition to simultaneously compute *all values* of a vectorial Boolean function $f : \{0,1\}^n \to \{0,1\}^m$. Since f may not be invertible (which is always the case if $n \neq m$), but quantum circuits must be invertible, f has to be tweaked. Given f, we define $F : \{0,1\}^{n+m} \to \{0,1\}^{n+m}$ by

$$F(x,y) = (x, y \oplus f(x)).$$

Obviously, F is invertible and $F^{-1} = F$. The corresponding unitary transformation on a system of $n + m$ qubits is given by

$$U_f(|x,y\rangle) = |x, y \oplus f(x)\rangle$$

(see Figure 13.7). Such a transformation can be efficiently implemented by combining elementary gates.

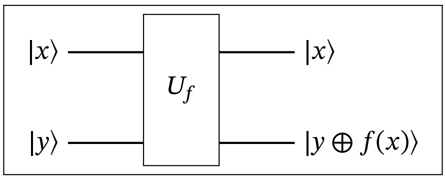


Figure 13.7. Unitary transformation associated to a Boolean function f.

Example 13.12. Let $f(x_1,x_2) = x_1x_2$ be the classical AND operation on two input bits (see Table 1.1). The corresponding invertible function is $F : \{0,1\}^3 \to \{0,1\}^3$, where $F(x_1,x_2,y) = (x_1,x_2,y \oplus x_1x_2)$. The quantum transformation is called a *Toffoli gate* and is given by

$$U_f(|x_1,x_2,y\rangle) = |x_1,x_2,y \oplus x_1x_2\rangle$$

on three input qubits with the basis states $|x_1\rangle$, $|x_2\rangle$ and $|y\rangle$. ◊

One can leverage F to compute f by setting the second input component y to the zero string 0^m. Then $F(x,0^m) = (x, f(x))$ and thus $U_f\,|x,0^m\rangle = |x, f(x)\rangle$. Furthermore, U_f maps a superposition $\sum_x a_x\,|x,0^m\rangle$ to

$$\sum_{x\in\{0,1\}^n} a_x\,|x, f(x)\rangle\,.$$

We combine the Walsh-Hadarmard transformation W (on the first n qubits) and U_f. This transforms the zero state into a superposition of all values of f (see Figure 13.8):

$$U_f(W\,|0^n\rangle, 0^m) = U_f\left(\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x, 0^m\rangle\right) = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x, f(x)\rangle.$$

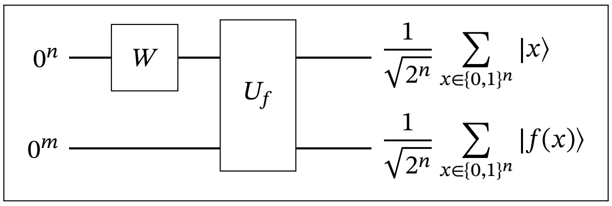

0^n
W
U_f
0^m
$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle$
$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|f(x)\rangle$

Figure 13.8. Input of the circuit is the zero state and output is the superposition of all values of f.

The simultaneous computation of *all* values of a function looks miraculous. However, the state is not directly accessible and a measurement only extracts a single value. Furthermore, the output is probabilistic, and values with a very small coefficient, i.e., a small probability, almost never occur. One therefore has to construct a clever quantum algorithm, which makes use of the superposition and outputs the requested value with a significant probability. One of the most important algorithms is the *Quantum Fourier Transform*, which we study in the following section.

Next, we explain the *Deutsch-Josza Algorithm* (see Figure 13.9). It illustrates the capabilities of quantum algorithms, although the underlying problem is of limited interest.

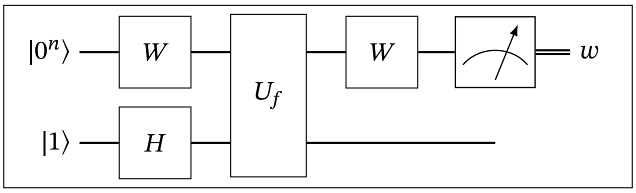


Figure 13.9. Deutsch-Josza algorithm. The measurement produces a classical binary string w of length n. If $w = 0^n$ is measured then f must be constant, otherwise, f is balanced.

Suppose a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is either *constant or balanced*, i.e., the number of input strings having the output 0 and 1 is equal. A classical non-probabilistic algorithm requires (in the worst case) $2^{n-1} + 1$ function calls in order to determine whether or not f is balanced. This is only feasible for small n. In contrast, the Deutsch-Josza quantum algorithm is polynomial in n. The first step of the Deutsch-Josza algorithm is to apply the Walsh-Hadamard transformation $H^{\otimes(n+1)}$ to the input state $|0^n, 1\rangle$. Since $H\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, we obtain the state

$$\begin{aligned}|\psi\rangle &= (H^{\otimes n} \otimes H)(|0\rangle \otimes \cdots \otimes |0\rangle \otimes |1\rangle) \\ &= H^{\otimes n}\,|0^n\rangle \otimes H\,|1\rangle \\ &= \frac{1}{\sqrt{2^{n+1}}} \sum_{x \in \{0,1\}^n} |x\rangle \otimes (|0\rangle - |1\rangle).\end{aligned}$$

Next, $U_f(|x, y\rangle) = |x, y \oplus f(x)\rangle$ is applied to the state ψ:

$$U_f\,|\psi\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x \in \{0,1\}^n} |x\rangle \otimes ((|0\rangle - |1\rangle) \oplus f(x)).$$

Suppose $|x\rangle$ is a basis state $|x_1 \dots x_n\rangle$. If $f(x) = 0$ then $|0\rangle - |1\rangle$ remains unchanged. If $f(x) = 1$ then $|0\rangle - |1\rangle$ is mapped to $|1\rangle - |0\rangle = -(|0\rangle - |1\rangle)$. In both cases, we have

$$(|0\rangle - |1\rangle) \oplus f(x) = (-1)^{f(x)}(|0\rangle - |1\rangle),$$

and thus obtain

$$U_f\,|\psi\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)}\,|x\rangle \otimes (|0\rangle - |1\rangle).$$

In the next step, $H^{\otimes n} \otimes Id$ is applied to $U_f\,|\psi\rangle$, so that $|x\rangle$ is mapped to $H^{\otimes n}\,|x\rangle$. One can check the expansion

$$H^{\otimes n}\,|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{z \in \{0,1\}^n} (-1)^{x \cdot z}\,|z\rangle$$

(see Exercise 8), where $x \cdot z$ is the dot product modulo 2. This yields

$$\left(H^{\otimes n} \otimes Id\right)(U_f\,|\psi\rangle) = \sum_{z \in \{0,1\}^n} \sum_{x \in \{0,1\}^n} \frac{1}{2^n}(-1)^{f(x)+x\cdot z}\,|z\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

Finally, we measure the first n qubits. The coefficient of the basis state $|0^n\rangle$ is

$$\sum_{x \in \{0,1\}^n} \frac{1}{2^n}(-1)^{f(x)},$$

since $z = 0^n$ obviously yields $x \cdot z = 0$ for all $x \in \{0,1\}^n$. If f is constant, then the above coefficient is either 1 or -1. Hence the probability of measuring 0^n is equal to 1 and any measurement must give 0^n. On the other hand, if f is balanced then positive and negative terms cancel and the probability of measuring 0^n is 0. The Deutsch-Josza algorithm outputs *f is constant*, if a measurement gives 0^n, and otherwise *f is balanced*. This result is always correct and the quantum algorithm runs in polynomial time.

13.4. Quantum Fourier Transform

The Deutsch-Josza algorithm demonstrates that measuring a quantum state can provide useful information. The *Quantum Fourier Transform* is a key algorithm, which efficiently finds a period of a large input vector. In Section 13.5 below we will see how this can be leveraged to solve the factoring problem.

The classical *Discrete Fourier Transform* (DFT) is a bijective $\mathbb{C}$-linear map on $\mathbb{C}^N$. A sequence of N complex numbers (in the discrete time domain) is mapped to the frequency domain. The resulting vector of complex Fourier coefficients reveals the periodic structure of the input data.

Suppose the input vector is the complex vector $(a_0, a_1, \dots, a_{N-1})$. Then the DFT is $(y_0, y_1, \dots, y_{N-1})$ and the Fourier coefficients are defined by

$$y_k = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} a_x e^{-2\pi i x k/N}, \text{ where } k = 0, 1, \dots, N-1.$$

We note that the above unitary DFT differs from other variants by a normalization factor. The computation can be accelerated using the Fast Fourier Transform (FFT). The DFT is given by the following unitary $N \times N$ matrix:

$$U = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ & & & \dots & \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix},$$

where $\omega = e^{-2\pi i/N}$ is a primitive N-th root of unity.

The original data can be recovered from the Fourier coefficients using the *inverse DFT*:

$$a_x = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} y_k e^{2\pi i x k/N}, \text{ where } x = 0, 1, \dots, N-1.$$

The corresponding matrix of the inverse DFT is $U^{-1} = \overline{U}^T$, the conjugate transpose matrix.

The DFT computes the discrete spectrum of the input data. If the data of length N is r-periodic and N is divisible by r, then the Fourier coefficients y_k are nonzero only for multiples of $\frac{N}{r}$. In general, a Fourier amplitude $|y_k| \gg 0$ indicates that $\frac{N}{k}$ is an *approximate multiple* of the period.

Example 13.13. Let $a = (1,2,1,2)^T$ be a data vector of length $N = 4$. The data is $r = 2$-periodic. The unitary 4×4 matrix that describes the DFT is

$$U = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

We compute the Fourier coefficients $y = Ua = (3,0,-1,0)^T$. Only the coefficients y_0 and y_2 are nonzero and hence a is $\frac{4}{2} = 2$-periodic. The input vector a can be recovered from the Fourier coefficients by computing $a = U^{-1}y = \overline{U}^T y$. ◊

In a quantum setting, we want to compute the DFT of a large input vector $(a_0, a_1, \dots, a_{N-1})$ of length $N = 2^s$ and find a *hidden period* of the data. The basic idea is that the indices k with Fourier coefficients $|y_k|^2 \gg 0$ reveal the period. The measurement of a state vector of Fourier amplitudes will give such indices k with a significant probability.

The Quantum Fourier Transform (QFT) does a DFT on the amplitudes of the quantum state and outputs a superposition of Fourier coefficients:

$$|\psi\rangle = C \sum_{x=0}^{N-1} a_x |x\rangle \mapsto U|\psi\rangle = C \sum_{k=0}^{N-1} y_k |k\rangle .$$

The scaling factor C ensures that the coefficient vector of $|\psi\rangle$ is normalized. The Fourier coefficients are

$$y_k = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} a_x e^{2\pi i x k / N}.$$

Note that the above QFT uses the primitive N-th root of unity $\omega = e^{2\pi i/N}$ and not the conjugate value $e^{-2\pi i/N}$. One can show that the QFT has an efficient circuit and runs in time $O(\text{size}(N)^2)$. In the simplest case ($N = 2$), the QFT is given by a single Hadamard gate.

Now, suppose that the input vector $(a_0, a_1, \dots, a_{N-1})$ is r-periodic, where r is an unknown period between 1 and N. One prepares the state $|\psi\rangle = C\sum_{x=0}^{N-1} a_x |x\rangle$ of an n-qubit system, applies the QFT and measures the state $U|\psi\rangle = C\sum_{k=0}^{N-1} y_k |k\rangle$. With a high probability, the measured output k is an approximate multiple of $\frac{N}{r}$.

13.5. Shor's Factoring Algorithm

The assumption that factoring is a hard problem plays an important role in public-key cryptography, and especially for RSA encryption and signature schemes. The most efficient classical factoring algorithm is the *Number Field Sieve*, which has subexponential complexity (see Section 9.7). In 1994, Peter Shor found a quantum algorithm that runs in polynomial time ([**Sho94**]). This algorithm is a major application of quantum computing. It has been successfully implemented for toy examples (like factoring $21 = 7 \cdot 3$) and will certainly be applied to real-world problems as soon as quantum computers with thousands of qubits become available.

Shor's algorithm finds a *hidden period* of a function and is based on the Quantum Fourier Transform.

Firstly, we explain how to derive the unknown factors of a composite number n from the multiplicative order of an element $a \in \mathbb{Z}_n^*$. Note that the multiplicative order of a is also the least period of the function $f(x) = a^x \mod n$.

Suppose that $n = pq$, where n is known and p, q are unknown. Choose a uniform random integer a with $1 < a < n$. If $\gcd(a, n) \neq 1$, then $\gcd(a, n)$ is either p or q and the unknown factors are found. However, the probability of $\gcd(a, n) \neq 1$ is very small if the prime factors are large and a is randomly chosen.

Now assume that $\gcd(a, n) = 1$; then $a \mod n \in \mathbb{Z}_n^*$ and

$$r = \operatorname{ord}(a) \mid \operatorname{ord}(\mathbb{Z}_n^*) = (p-1)(q-1)$$

(see Corollary 4.14). By definition, $a^r \equiv 1 \mod n$. If r is even, then

$$a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \mod n,$$

and thus $n \mid (a^{r/2} - 1)(a^{r/2} + 1)$. Since $\operatorname{ord}(a) \neq \frac{r}{2}$, we have $n \nmid (a^{r/2} - 1)$ and there are two possibilities:

(1) p divides one of the two factors and q divides the other. In this case $\gcd(a^{r/2}+1, n)$ gives p or q.

(2) $n \mid (a^{r/2} + 1)$, then the algorithm fails and one has to choose another base a.

The algorithm is successful if r is even and $n \nmid (a^{r/2} + 1)$. Closer analysis shows that the probability of success is at least 50% (compare [**NC00**]): let r_p and r_q be the order of a in $\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$, respectively. Then r is odd if and only if r_p and r_q are odd (see Exercise 10). Now suppose that r is even and 2^d is the maximal power of 2 that divides r. We have $a^{r/2} + 1 \equiv 0 \mod n$ if and only if $a^{r/2} \equiv -1 \mod p$ and $a^{r/2} \equiv -1 \mod q$. This requires $r_p \nmid \frac{r}{2}$ and $r_q \nmid \frac{r}{2}$. Since $r_p \mid r$ and $r_q \mid r$, we obtain $2^d \mid r_p$ and $2^d \mid r_q$.

Summarizing, the algorithm fails if either $2 \nmid r_p$ and $2 \nmid r_q$ or $2^d \mid r_p$ and $2^d \mid r_q$. If a is chosen uniformly at random, then the probability for this to happen is at most 50%.

Example 13.14. Let $n = 77$ and $a = 3$; then $\gcd(a, n) = 1$. Suppose we know the order of $a \mod n$ in the multiplicative group $\mathbb{Z}_n^*$: the order is $r = \operatorname{ord}(3 \mod 77) = 30$ and this is an even number. We obtain

$$a^{r/2} + 1 = 3^{15} + 1 \equiv 35 \mod 77.$$

We compute $\gcd(a^{r/2} + 1, n) = \gcd(35, 77) = 7$ and obtain one of prime factors of $n = 77$. Note that $\gcd(a^{r/2} - 1, n) = \gcd(33, 77) = 11$ gives the other factor of n.

In fact, we succeeded in this example since $r_p = \text{ord}(3 \mod 7) = 6$ is even and $r_q = \text{ord}(3 \mod 11) = 5$ is odd. ◊

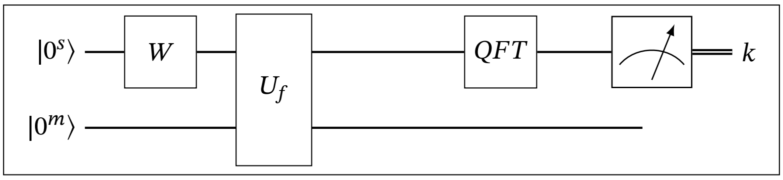


Figure 13.10. Shor's algorithm uses the Quantum Fourier Transform and finds a hidden period of a function f.

Now, the quantum part of Shor's algorithm is to compute the unknown order r of a given residue class $a \in \mathbb{Z}_n^*$. For that purpose, one prepares a superposition of input values $x = 0, 1, \ldots, N-1$ and simultaneously computes all $a^x \mod n$. The values are r-periodic, i.e., $a^x \equiv a^{x+r}$. The QFT is applied to the state and we will see that a measurement reveals the period with high probability. The sequence must be significantly longer than the period and it turns out that $N = 2^s$ with $n^2 \leq N \leq 2n^2$ is a reasonable choice.

We view $f(x) = a^x \mod n$ as a Boolean function. The corresponding unitary transformation on quantum bits is $U_f\,|x, y\rangle = |x, y \oplus f(x)\rangle$. The first register has s qubits and the second has $m = \text{size}(n)$ qubits.

The Walsh-Hadamard transformation maps $|x\rangle = |0^s\rangle$ to a superposition of all basis states. We set $|y\rangle = |0^m\rangle$, apply U_f and obtain the state

$$|\psi\rangle = U_f(W\,|0^s\rangle, |0^m\rangle) = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x, f(x)\rangle .$$

Next, the QFT operator U is applied to the first register while the second remains unchanged:

$$(U \otimes Id)\,|\psi\rangle = \frac{1}{N} \sum_{x=0}^{N-1} \sum_{k=0}^{N-1} e^{2\pi i x k/N}\,|k, f(x)\rangle .$$

Finally, the first register is measured (see Figure 13.10). The second register takes a random value $u \in im(f)$ and the probability of measuring $|k, u\rangle$ is

$$\left| \frac{1}{N} \sum_{x:\ f(x)=u} e^{2\pi i x k/N} \right|^2 .$$

There is a high probability that k is an approximate multiple of $\frac{N}{r}$ and the inequalities

$$\left|k - j\frac{N}{r}\right| < \frac{1}{2} \Rightarrow \left|\frac{k}{N} - \frac{j}{r}\right| < \frac{1}{2N} \Rightarrow \left|\frac{k}{N} - \frac{j}{r}\right| < \frac{1}{2n^2}$$

hold for some j. One can show that at most one fraction $\frac{j}{r}$ with $0 < j < n$ and $0 < r < n$ can satisfy this inequality. The fraction and the requested period r can be efficiently determined using the *continued fraction expansion* (see Example 13.16 below).

The reader might be surprised that the QFT of the *first register* gives anything interesting, since the amplitudes of the first register of $|\psi\rangle$ are constant. However, the amplitudes are partitioned by the second register, i.e., by different values of $u = f(x) = a^x \mod n$. We can rewrite $|\psi\rangle$ as

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{u \in im(f)} \sum_{f(x)=u} |x, u\rangle .$$

The amplitudes with different u in the second register do not interfere with each other when the QFT is applied to the first register. Now, for a fixed second register u, the first register is r-periodic and applying the QFT gives peaks at multiples of $\frac{N}{r}$. If N is divisible by r, the Fourier amplitudes are zero outside multiples of $\frac{N}{r}$.

Remark 13.15. Shor's algorithm requires around $3\,\mathrm{size}(n)$ qubits and uses $O(\mathrm{size}(n)^3)$ operations, and optimizations are known.

Example 13.16. We continue Example 13.14 and apply Shor's algorithm to $n = 77$ and $a = 3 \in \mathbb{Z}_n^*$. We want to compute $\mathrm{ord}(a)$. First, we have to define $N = 2^s$. Since $n^2 = 5929$ and $2n^2 = 11858$, we choose $s = 13$ and $N = 8192$. We create a superposition of the basis states $|x, 3^x \mod 77\rangle$ for $x = 0, 1, \dots, 8191$ and obtain

$$|\psi\rangle = \frac{1}{\sqrt{8192}} \sum_{x=0}^{8191} |x,\, 3^x \mod 77\rangle .$$

Note that we need 13 qubits for the first register and 7 qubits for the second. We can reorder the terms with respect to the second register $u = 3^x \mod 77$. Suppose for example that $u = 59$; then the corresponding terms in $|\psi\rangle$ are

$$\frac{1}{\sqrt{8192}} \left(|19, 59\rangle + |49, 59\rangle + |79, 59\rangle + \cdots + |8179, 59\rangle\right).$$

The expansions for other values u in the second register look similar and the period $r = 30$ is clearly visible, but the state is not directly accessible to an observer. Instead, we apply the QFT to the first register and measure it. The second register takes a random value $u \in im(f)$, for example $u = 59$. The amplitudes $|y_k|$ of $|k, u\rangle$ for $k = 0, 1, \dots, 8191$ and $u = 59$ are shown in Figure 13.11. The squares $|y_k|^2$ give the probability that $|k, 59\rangle$ is measured. The probabilities for any other value $u \in im(f)$ are identical. Closer inspection shows peaks at all multiples of $\frac{N}{r} \approx 273$. Table 13.1 lists some amplitudes around $k = 273$.

Again, we remark that these amplitudes are not accessible to an observer, but the measured value k is likely to be a multiple of $\frac{N}{r}$. Suppose that $k = 7100$. We expect

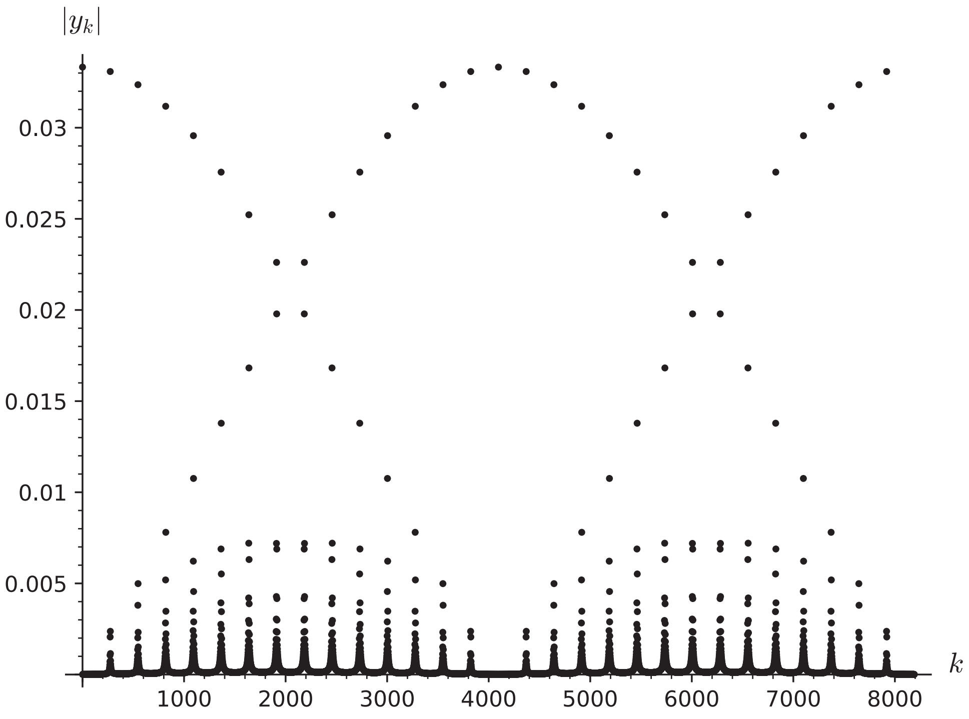


Figure 13.11. Fourier amplitudes of $|k, 59\rangle$. The spectrum has peaks at multiples of $\frac{8192}{30} \approx 273$.

Table 13.1. Absolute values of selected amplitudes.

k	270	271	272	273	274	275
$\lvert y_k \rvert$	0.00071	0.00106	0.002060	0.033082	0.002372	0.001149

that $\frac{k}{N} = \frac{7100}{8192}$ is close to a fraction $\frac{j}{r}$ where j and r are integers less than $n = 77$. In this toy example, we could simply try out the possible values for j and r, but in general, the efficient method of *continued fractions expansions* is used.

The idea is to approximate a real number x by continued fractions of integer numbers. The number is split into its integer part $\lfloor x \rfloor$ and its fractional part ϵ_0:

$$x = \lfloor x \rfloor + \epsilon_0 = \lfloor x \rfloor + \frac{1}{\left(\frac{1}{\epsilon_0}\right)}.$$

Next, $\frac{1}{\epsilon_0}$ is split into an integer and a fractional part and we obtain

$$x = \lfloor x \rfloor + \epsilon_0 = \lfloor x \rfloor + \frac{1}{\lfloor \frac{1}{\epsilon_0} \rfloor + \epsilon_1}.$$

We continue in the same fashion, write $\epsilon_1 = \frac{1}{\left(\frac{1}{\epsilon_1}\right)}$ and split $\frac{1}{\epsilon_1}$ into an integer and a fractional part. The method terminates after a finite number of steps if x is a rational number, and otherwise approximates x.

For our example, we let SageMath compute the sequence of integer parts:

```
sage: (7100/8192).continued_fraction()
[0; 1, 6, 1, 1, 136]
```

Hence the complete continued fractions expansion is

$$\frac{7100}{8192} = 0 + \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{1 + \frac{1}{136}}}}}.$$

However, we need an approximation with a denominator less than $n = 77$ and so we discard the last fraction $\frac{1}{136}$. Then

$$\frac{7100}{8192} \approx \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{1 + \frac{1}{1}}}} = \frac{13}{15}.$$

We obtain $\frac{j}{r} = \frac{13}{15}$ and therefore the period must be a multiple of 15 less than $n = 77$.

One checks that $3^{15} \equiv 34 \not\equiv 1 \mod 77$ and $3^{30} \equiv 1 \mod 77$. We conclude that $r = \text{ord}(3) = 30$. Finally, we verify that $\frac{N}{r} = \frac{8192}{30} \approx 273.07$, which explains the peaks of the spectrum at multiples of this number. ◊

We have seen above that Shor's algorithm can efficiently solve the factoring problem. The other major cryptographic problem, the *discrete logarithm problem*, can also be solved with a period-finding algorithm. This can be applied to the multiplicative group of integers modulo a prime number p and also to the group of points on an elliptic curve over a finite field.

Suppose that $A = g^a$ holds in a cyclic group $G = \langle g \rangle$ of order n and a is unknown. Consider the map

$$f : \mathbb{Z} \times \mathbb{Z} \to G, \ f(x, y) = A^x g^{-y}.$$

This map is $(1, a)$-periodic since

$$f(x + 1, y + a) = A^{x+1} g^{-y-a} = g^{ax+a} g^{-y-a} = A^x g^{-y} = f(x, y).$$

Any period (r,s) is a multiple of $(1,a) \mod n$, since

$$f(x+r,\, y+s) = f(x,y) \Longrightarrow A^{x+r}g^{-y-s} = A^x g^{-y}.$$

This implies $a(x+r) - y - s \equiv ax - y \mod n$ and hence $ar \equiv s \mod n$. If we find an approximate period (r,s) using an efficient quantum algorithm, then we can also compute the discrete logarithm a.

This means that *classical public-key cryptography is broken* once sufficiently large and stable quantum computers become available!

On the other hand, it is expected that symmetric algorithms are less severely affected by the capabilities of quantum computers. *Grover's algorithm* [**Gro96**] provides a speedup from n to $\sqrt{n}$ evaluations, which effectively halves the security level. As a consequence, AES with 128-bit keys provides only 64 bits of post-quantum security, but attacking 256-bit AES with 128-bit post-quantum security is still unfeasible.

13.6. Quantum Key Distribution

A remarkable cryptographic application of quantum mechanics is *key distribution* over an insecure channel. There are popular, well-known key establishment protocols, for example the Diffie-Hellman key exchange (see Section 10.3), but once large-scale quantum computing becomes available, the conventional public-key protocols are broken.

Quantum Key Distribution (QKD) is using *qubits* to transfer a secret key. Unlike the quantum algorithms discussed above, QKD does not require an n-qubit system, but uses a sequence of single qubits or pairs of entangled qubits. This is much easier to realize and QKD has already been successfully implemented in practice, where polarized photons were sent over large distances.

In this section, we present the BB84 (Bennett and Brassard) quantum key distribution protocol [**BB84**]. Other protocols (in particular Ekert's E91 protocol [**Eke91**]) are also used in practical implementations.

The idea of QKD is that quantum bits cannot be measured without changing the state. Furthermore, an unknown quantum state cannot be cloned, since the internal state is unknown to an observer who does not know how the system was initialized.

Suppose that Alice and Bob want to exchange a key of length n and they have access to a public communication channel and a quantum channel of single polarized photons. They use two different bases B_0 and B_1 (polarizations) of the single-qubit space $\mathbb{C}^2$. Assume that the standard basis is

$$B_0 = \{|0\rangle, |1\rangle\}.$$

Applying the Hadamard transformation yields the Hadamard basis B_1 which is diagonal to B_0:

$$B_1 = HB_0 = \left\{ \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle,\, \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle \right\} = \{|+\rangle, |-\rangle\}.$$

Let δ be a positive integer constant which determines the probability of success of the protocol. Alice chooses a uniform random key string k of length $4n + \delta$ bits. Furthermore, Alice and Bob choose $4n + \delta$ bits that will determine whether B_0 or B_1 is used:

$$k \overset{\$}{\leftarrow} \{0,1\}^{4n+\delta},\ b_A \overset{\$}{\leftarrow} \{0,1\}^{4n+\delta},\ b_B \overset{\$}{\leftarrow} \{0,1\}^{4n+\delta}.$$

Now Alice sends the key k as a sequence of single qubits to Bob. She uses the basis B_0 if the corresponding bit of b_A is 0, or otherwise B_1.

Bob receives the sequence of qubits and measures them with respect to B_0 or B_1, depending on the corresponding bit of b_B. If the i-th bit of b_A and b_B are equal, then Bob measures the correct i-th bit of k. Otherwise, the probability of a correct key bit is only 50%. For example, suppose that Alice is using B_0 and sending the bit 1; then the transmitted qubit is $|1\rangle$. If Bob chooses the basis B_1, then

$$|1\rangle = \frac{1}{\sqrt{2}}\,|+\rangle - \frac{1}{\sqrt{2}}\,|-\rangle\,.$$

Hence the probability of measuring $|-\rangle$, which corresponds to the bit 1, is only 50%.

After Bob has received and measured the key bits, both partners exchange their bases b_A and b_B via the conventional public channel. They discard the key bits that were measured in a different basis and restart the key exchange if less than $2n$ bits remain. Obviously, the constant δ determines the probability of a successful exchange. They keep the first $2n$ key bits. The following step aims to reveal the interference of an adversary. Alice chooses a subset of n key bits and sends Bob the selected positions. They exchange the associated key bits via the public channel. They compare the bits and abort the protocol if the number of errors is higher than expected. The remaining n bits are used as a secret key, which needs to be further transformed (*information reconciliation* and *privacy amplification*), in order to reduce the effects of errors and undetected interference by adversaries.

We are now discussing the security of the BB84 protocol. Firstly, the non-quantum communication channel between Alice and Bob can be public, but integrity is important. Furthermore, Alice has to generate and transmit *single* qubits, for example single polarized photons, since an eavesdropper could otherwise use any extra particles with the same state for a measurement.

It may seem surprising that the key can be sent without any protection. However, an eavesdropper has to measure the qubits in order to get any information. This requires choosing a basis, i.e., B_0 or B_1, which is incorrect in about half of the cases. Remember that the correct basis is only known to Alice during the transmission of the qubits. At first, Alice and Bob are not aware of an interception, but the error rate of the check bits will increase significantly. In fact, Bob will measure around 25% incorrect check bits, since the error rate is 50% if an eavesdropper used the wrong basis. Therefore, Alice and Bob can detect an adversary who has intercepted a sufficient number of quantum bits. Assuming that Alice and Bob accept a maximum error rate of 2.5%, an adversary can only eavesdrop around 10% of the bits if they want to remain undetected. Privacy amplification methods reduce an adversary's partial information on the key by producing a new, shorter key.

Table 13.2. Quantum key distribution example.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Alice's key	0	1	0	0	1	1	0	1	0	0	1	0	1	0	1	1	0
Alice's basis	1	0	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0
Alice sends	+	1	+	+	1	1	0	−	+	+	1	0	1	+	−	−	0
Bob's basis	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	0	1
Bob measures	+	1	1	0	1	−	0	−	1	+	−	+	+	+	−	0	+
Same basis	✓	✓			✓		✓	✓		✓				✓	✓		
Shared key	0	1			1		0	1		0				0	1		
Check bits		1			1					0				0			
Key bits	0						0	1							1		

Example 13.17. (See Table 13.2) Suppose $n = 4$ and $\delta = 1$. Alice generates the random key

$$k = 0100\ 1101\ 0010\ 1011\ 0$$

of length 17. Alice and Bob's choice of bases is given by the binary strings

$$b_A = 1011\ 0001\ 1100\ 0111\ 0,$$
$$b_B = 1000\ 0101\ 0111\ 1110\ 1,$$

where 0 represents the basis $B_0 = \{|0\rangle, |1\rangle\}$ and 1 the basis $B_1 = \{|+\rangle, |-\rangle\}$. Alice sends the following qubits:

$$+1 + +110 - + + 101 + - - 0.$$

Alice and Bob exchange b_A and b_B. The following positions coincide: 1, 2, 5, 7, 8, 10, 14, 15. Hence Alice and Bob used the same basis for these positions and 8 shared key bits remain. Alice chooses positions 2, 5, 10, 14 to check for eavesdropping, which would probably change at least one bit. They exchange the check bits and verify them. If the check bits match, then they accept the key exchange. The resulting key 0011 is defined by the remaining four positions 1, 7, 8 and 15. ◊

13.7. Summary

- Quantum computing relies on quantum mechanics and uses quantum bits (qubits) instead of classical bits.
- The state of a qubit is a superposition of the basis states $|0\rangle$ and $|1\rangle$.
- A single qubit state can be represented by a point on the Bloch sphere.
- The measurement of a qubit gives a classical bit. The probability of measuring 0 or 1 is given by the square of the amplitude of $|0\rangle$ and $|1\rangle$, respectively.
- A system of n qubits has 2^n basis states and a measurement gives n classical bits.
- A quantum gate has n input qubits and n output qubits and the transformation is described by a unitary matrix. Quantum algorithms use quantum gates and measurements.
- The Quantum Fourier Transform (QFT) maps an input state to a superposition of Fourier coefficients.
- Shor's factoring algorithm leverages the QFT to solve the factoring problem in polynomial time. Quantum computers can break conventional public-key cryptography, but sufficiently large and stable systems are not yet available.
- The Quantum Key Distribution (QKD) protocol BB84 uses a sequence of single qubits for key distribution. Since the internal state of a qubit is changed by a measurement, the interference by an adversary can be detected.

Exercises

1. Show that the Bell state $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ cannot be written as the product of two single qubit states $(a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle)$.
2. Show that applying the CNOT gate to $H|0\rangle \otimes |0\rangle$ gives the above Bell state. Depict the corresponding circuit. Give the other three Bell states $H|0\rangle \otimes |1\rangle$, $H|1\rangle \otimes |0\rangle$ and $H|1\rangle \otimes |1\rangle$.
3. Describe the transformations of Pauli-X, Y, Z, S (Phase), T $(\frac{\pi}{8})$ and H gates on the Bloch sphere.
4. Prove that there is a bijection between the state space for a single-qubit system and the complex projective space $\mathbb{P}^1(\mathbb{C})$.
5. Give the matrix of the Walsh-Hadamard transformation of two qubits.
6. Determine the matrix of the Toffoli gate on three input qubits.
7. Show that the Toffoli gate gives a quantum analogue of the classical NAND operation.
8. Let $x \in \{0,1\}^n$. Prove the formula

$$H^{\otimes n} |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{z \in \{0,1\}^n} (-1)^{x \cdot z} |z\rangle .$$

9. Give an explicit description of the Quantum Fourier Transform for $N = 2$ and $N = 4$.
10. Suppose that $n = pq$, where p and q are prime numbers. Let $a \in \mathbb{Z}_n^*$ and let r, r_p, r_q be the order of a in $\mathbb{Z}_n^*$, $\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$, respectively. Show that r is the least common multiple of r_p and r_q.
11. Let $n = 47053$ and $a = 11 \in \mathbb{Z}_n^*$. Suppose that $\operatorname{ord}(a) = 7770$ is already known. Find the factors p and q of n.
12. Alice and Bob use the BB84 protocol and Mallory intercepts m key bits. Determine the expected number of faulty bits, i.e., bits that differ from Alice's original key.
13. Suppose $n = 2048$ and $\delta = 128$ in the BB84 protocol. Mallory intercepts 64 key bits. How many of the check bits are likely to differ if no other transmission errors occur?

Chapter 14

Lattice-based Cryptography

The following two chapters deal with public-key cryptosystems based on lattices and codes, respectively. A key motivation is the emergence of quantum computers which are able to break RSA, Diffie-Hellman and elliptic curve cryptosystems. Fortunately, symmetric schemes like AES are less affected by the effects of quantum algorithms. Now the relatively new field of post-quantum cryptography studies encryption and signatures schemes which are believed to be secure in the presence of quantum computers. We focus on lattices and codes used in many proposals and look at encryption schemes.

This chapter introduces the basics of lattices and their applications in cryptography. Lattices are discrete subgroups of $\mathbb{R}^n$ and there are computational problems, for example finding the shortest vector in a lattice, which are believed to be hard. Solving a system of linear equations is easy, but solving a random system of noisy linear equations modulo an integer (learning with errors) is hard. Lattice-based cryptography offers strong security guarantees and is believed to resist quantum attacks.

We outline the fundamentals of lattices in Section 14.1. Lattice algorithms and in particular the LLL algorithm are studied in Section 14.2. The following Sections 14.3, 14.4 and 14.5 deal with the public-key encryption schemes GGH, NTRU and LWE, respectively. There are also promising lattice-based signature schemes, but they are not covered in this book.

We refer the reader to the textbooks [**HPS08**], [**Gal12**] and the articles [**MR09**], [**Pei14**] for additional reading on lattice-based cryptography.

14.1. Lattices

A lattice Λ is a *discrete subgroup* of $\mathbb{R}^n$. The *trivial lattice* is $\Lambda = \{0\}$ and a standard example is the lattice $\Lambda = \mathbb{Z}^n$ of vectors with integer coordinates. In mathematics, a lattice can also refer to a partially ordered set in which two elements have a least upper bound and a greatest lower bound, but this not used in our context.

Definition 14.1. A subset $\Lambda \subset \mathbb{R}^n$ is called *discrete* if every point $v \in \Lambda$ possesses an environment $U = \{w \in \mathbb{R}^n \mid ||v - w|| < \epsilon\}$, i.e., an open ball of radius $\epsilon > 0$, such that $U \cap \Lambda = \{v\}$, i.e., v is the only lattice point in U. A discrete subgroup of $\mathbb{R}^n$ is called a *lattice*. ♢

It follows from the above definition that every bounded set and in particular every ball $\{v \in \mathbb{R}^n \mid ||v|| < d\}$ only contains a finite number of lattice points. All nontrivial lattices are infinite sets, but they have a *finite basis*.

Let $V \subset \mathbb{R}^n$ be the real vector space generated by Λ. We define the *rank* of Λ to be the *dimension* of V. The following Proposition shows that a lattice Λ of rank r has a $\mathbb{Z}$-basis $\{v_1, \dots, v_r\} \subset \Lambda$.

Proposition 14.2. *Let Λ be a nontrivial lattice of rank r. Then there is a basis $B = \{v_1, v_2, \dots, v_r\} \subset \Lambda$, i.e., a set of linearly independent vectors such that*

$$\Lambda = \{x_1v_1 + x_2v_2 + \cdots + x_rv_r \mid x_1, x_2, \dots, x_r \in \mathbb{Z}\}.$$

Proof. Our proof follows [**Gal12**]. Let $B = \{v_1, v_2, \dots, v_r\}$ be a set of linearly independent vectors in Λ. We want to transform B into a basis B' of Λ. For $d \le r$ we let V_d be the real vector space generated by $v_1, \dots, v_d$. The lattice $\Lambda_d = V_d \cap \Lambda$ has rank $d \le r$ and $\Lambda_r = \Lambda$. We now prove the claim by induction. For $d = 1$, we can easily find a basis of Λ_1: we replace v_1 by the shortest nonzero multiple $v_1' = \alpha v_1$ such that $v_1' \in \Lambda_1$. Now we assume that Λ_{d-1} has a basis $\{v_1', \dots, v_{d-1}'\}$. Consider the bounded and discrete set

$$S = \Lambda_d \cap \{\alpha_1 v_1' + \cdots + \alpha_{d-1}v_{d-1}' + \alpha_d v_d \mid \alpha_1, \dots, \alpha_{d-1} \in [0,1[\text{ and } \alpha_d \in [0,1]\}.$$

Let $v_d' = \alpha_1 v_1' + \cdots + \alpha_{d-1}v_{d-1}' + \alpha_d v_d$ be the element in S with smallest nonzero coefficient α_d. It is obvious that $B' = \{v_1', \dots, v_{d-1}', v_d'\}$ is linearly independent and it remains to show that B' is a basis of Λ_d. To this end, given any vector $v = \beta_1 v_1' + \cdots + \beta_{d-1}v_{d-1}' + \beta_d v_d \in \Lambda_d$, there are integer coefficients $x_i \in \mathbb{Z}$ such that

$$w = v - x_1v_1' - \cdots - x_{d-1}v_{d-1}' - x_d v_d' \in S$$

and $\beta_d - x_d\alpha_d \in [0, \alpha_d[$. Note that $\beta_d - x_d\alpha_d$ is the coefficient of v_d in w. Since v_d' is the element in S with the smallest nonzero coefficient of v_d, we obtain $\beta_d - x_d\alpha_d = 0$ and hence $w \in \Lambda_{d-1}$. This shows that $v \in \Lambda_d$. □

In this chapter *we assume that the rank is maximal*, i.e., $\Lambda \subset \mathbb{R}^n$ and $rk(\Lambda) = n$, as the more general case is not substantially different. We write lattice vectors as *columns*, but row vectors are also used in the literature. Writing the basis vectors into

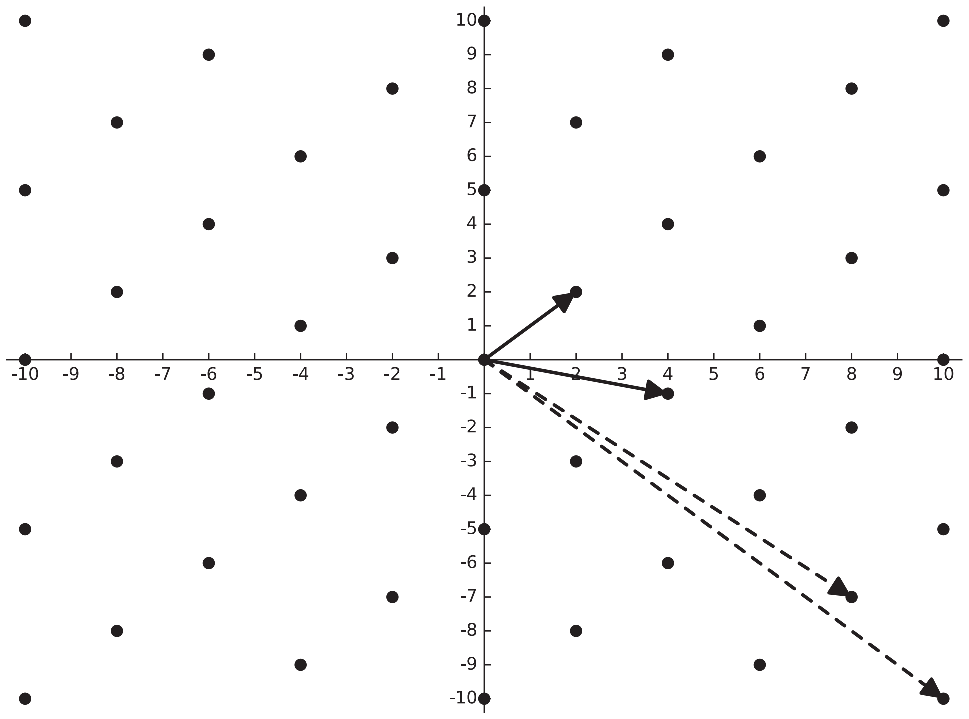


Figure 14.1. Lattice of dimension 2 and two different bases.

the columns defines a regular $n \times n$ matrix. By abuse of notation, we will use the same letter for a basis and the associated $n \times n$ matrix of column vectors. Two bases $B_1 = \{v_1, v_2, \dots, v_n\}$ and $B_2 = \{w_1, w_2, \dots, w_n\}$ of the same lattice Λ are connected by a unimodular $n \times n$ matrix U over $\mathbb{Z}$:

$$B_2 = B_1 U.$$

A matrix U is called *unimodular* if all entries are integers and $\det(U) = \pm 1$. Indeed, for each w_i there are $x_1, \dots, x_n \in \mathbb{Z}$ such that $w_i = x_1 v_1 + \cdots + x_n v_n$ and the coefficients $x_1, \dots, x_n$ form the i-th column of U. Conversely, each v_i can be represented by an integer linear combination of $w_1, \dots, w_n$. Therefore, we have $B_1 = B_2 \widetilde{U}$ for some integer matrix $\widetilde{U}$, from which we conclude that U is invertible, $\widetilde{U} = U^{-1}$ and $\det(U) = \pm 1$.

Example 14.3. Let $B_1 = \left\{ \begin{pmatrix} 4 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix} \right\}$ and assume that Λ is a lattice generated by B_1. The lattice is depicted in Figure 14.1, where B_1 is shown with continuous lines.

$B_2 = \left\{ \begin{pmatrix} 8 \\ -7 \end{pmatrix}, \begin{pmatrix} 10 \\ -10 \end{pmatrix} \right\}$ is another basis of Λ, and the basis B_2 is shown with dashed lines in Figure 14.1. One has $B_2 = B_1 U$, where U is the unimodular matrix

$$U = \begin{pmatrix} 3 & 4 \\ -2 & -3 \end{pmatrix}.$$

Intuitively, the first basis is 'better' than the second, since the vectors are shorter and closer to being orthogonal.

Definition 14.4. Let Λ be a lattice and B any basis of Λ. Then the *determinant* of Λ is defined by the absolute value

$$\det(\Lambda) = |\det(B)|.$$

The determinant of Λ does not depend on the chosen basis. ♢

A basis B of a lattice Λ defines a *fundamental parallelepiped*

$$P = \{x_1 v_1 + x_2 v_2 + \cdots + x_r v_r \mid x_1, x_2, \dots, x_r \in [0, 1[\ \}.$$

The determinant of a lattice can be described geometrically by the n-dimensional *volume* of the parallelepiped spanned by the basis vectors. We call this the *covolume* of Λ. In the literature, this is also called the *volume* of Λ.

Example 14.5. Consider the lattice and bases of Example 14.3. The covolume of Λ is

$$\det(\Lambda) = |\det(B_1)| = |\det(B_2)| = 10.$$

Definition 14.6. Let Λ be a lattice of dimension n. The dual lattice $\Lambda^* \subset \mathbb{R}^n$ is defined by the set of $y \in \mathbb{R}^n$ such that the dot product satisfies $x \cdot y \in \mathbb{Z}$ for all $x \in \Lambda$. ♢

Here we give a description of the dual lattice in terms of matrices. Remember that we assumed that our lattices have full rank. Let B be a basis of Λ. We have $y \in \Lambda^*$ if and only if $B^T y \in \mathbb{Z}^n$ or, equivalently, $y = (B^T)^{-1} x$ for some $x \in \mathbb{Z}^n$. This implies that the dual lattice is generated by the columns of $(B^T)^{-1}$. Furthermore, it follows that

$$\det(\Lambda^*) = \det |(B^T)^{-1}| = \frac{1}{|\det(B)|} = \frac{1}{\det(\Lambda)}.$$

Example 14.7. Consider the lattice Λ in Example 14.3. The *dual lattice* Λ^* is given by the columns of

$$(B_1^T)^{-1} = \begin{pmatrix} \frac{1}{5} & \frac{1}{10} \\ -\frac{1}{5} & \frac{2}{5} \end{pmatrix}.$$

The covolume of Λ^* is $\frac{1}{\det(\Lambda)} = \frac{1}{10}$. ♢

For cryptographic applications, one usually considers *q-ary lattices*, which are defined by integers and modular congruences.

Definition 14.8. Let $q \in \mathbb{Z}$ be a nonzero integer. A lattice Λ of dimension n is called *q-ary* if $q\mathbb{Z}^n \subset \Lambda \subset \mathbb{Z}^n$. ♢

Below, we suppose that Λ is an integer lattice, i.e., $\Lambda \subset \mathbb{Z}^n$. Hence Λ is q-ary if and only if the multiples $qe_1, \dots, qe_n$ of the unit vectors lie in Λ. Note that any integer lattice Λ is q-ary for some $q \in \mathbb{Z}$.

Example 14.9. The lattice Λ in Example 14.3 is 10-ary since $10e_1, 10e_2 \in \Lambda$. The lattice is q-ary for any integer multiple q of 10. ♢

q-ary lattices can be defined by any $m \times n$ matrix A over $\mathbb{Z}_q$:

$$\Lambda_q(A) := \{y \in \mathbb{Z}^n \mid y \equiv A^T x \mod q \text{ for } x \in \mathbb{Z}_q^m\}.$$

The n-dimensional lattice $\Lambda_q(A)$ is defined by the rows of A and $q\mathbb{Z}^n$. Note that here we have linear combinations of *rows* of A, not columns of A, as above. Furthermore, the kernel of A defines a lattice:

$$\Lambda_q^\perp(A) := \{y \in \mathbb{Z}^n \mid Ay \equiv 0 \mod q\}.$$

The lattices $\Lambda_q(A)$ and $\Lambda_q^\perp(A)$ have full rank since they contain $q\mathbb{Z}^n$.

$\Lambda_q(A)$ and $\Lambda_q^\perp(A)$ can also be viewed as *linear codes* over $\mathbb{Z}_q$, defined by the rows of A and the parity check matrix A, respectively (see Section 15.1). These two lattices are dual to each other, up to normalization (see Exercise 5):

$$\Lambda_q^\perp(A) = q\Lambda_q(A)^* \text{ and } \Lambda_q(A) = q\Lambda_q^\perp(A)^*.$$

Example 14.10. Consider the 10-ary lattice Λ from Example 14.3. The columns of B_1 are $(4, -1)$ and $(2, 2)$. Since $8 \cdot (4, -1) = (32, -8) \equiv (2, 2) \mod 10$, we discard the second vector and define the 1×2 matrix $A = (4 \ \ -1)$. Thus $\Lambda = \Lambda_{10}(A)$. The lattice $\Lambda_{10}^\perp(A)$ is defined by all solutions $(x, y) \in \mathbb{Z}^2$ of the modular equation $4x - y \equiv 0 \mod 10$. We have $\Lambda_{10}^\perp(A) = 10 \cdot \Lambda_{10}(A)^*$ and the lattice is defined by the columns of the matrix $\begin{pmatrix} 2 & 1 \\ -2 & 4 \end{pmatrix}$. ♢

Short lattice vectors play an important role in cryptographic applications.

Definition 14.11. Let Λ be a lattice and $\| \ \|$ a norm on $\mathbb{R}^n$ (usually the Euclidean norm).

(1) The norm of the shortest nonzero vector $v \in L$ defines $\lambda_1(\Lambda) := \|v\|$.
(2) The i-th successive minimum is defined by $\lambda_i(\Lambda) := \min_S(\max_{v \in S} \|v\|)$, where S runs over all linearly independent sets $S \subset \Lambda$ with $|S| = i$. ♢

We state several computational lattice problems:

Definition 14.12. Let Λ be a lattice and $\| \ \|$ a norm on $\mathbb{R}^n$ (usually the Euclidean norm).

(1) Shortest Vector Problem (SVP): Find a shortest nonzero vector $v \in \Lambda$.

(2) Shortest Independent Vector Problem (SIVP): Find linearly independent vectors $v_1, \dots, v_n$ in Λ such that $\max_i \|v_i\| = \lambda_n(\Lambda)$.
(3) Closest Vector Problem (CVP): given any target vector $w \in \mathbb{R}^n$, find the closest lattice point $v \in \Lambda$ to w. ♢

One also considers *approximation* variants of these problems. Let $\gamma \geq 1$. In SVP_γ, one has to find a vector v with $\|v\| \leq \gamma\, \lambda_1(\Lambda)$. Similarly, the SIVP_γ problem is to find linearly independent vectors $v_1, \dots, v_n$ such that $\max_i \|v_i\| \leq \gamma\, \lambda_n(\Lambda)$.

In CVP_γ, the goal is to find a vector v such that the distance to a target vector w is at most γ times the distance of the closest lattice vector to w.

Example 14.13. In Example 14.3 (see Figure 14.1), the shortest vectors are $v = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$ and $-v$, and thus $\lambda_1(\Lambda) = \sqrt{8}$. The basis B_1 is not a solution to SIVP, but instead the basis

$$\left\{ \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} -2 \\ 3 \end{pmatrix} \right\}.$$

Hence $\lambda_2(\Lambda) = \sqrt{13}$. However, B_1 is a solution to SIVP_γ for $\gamma \geq \frac{\sqrt{17}}{\sqrt{13}}$ since $\lambda_2(B_1) = \sqrt{17}$.

The closest lattice vector to the target $w = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$ is $v = \begin{pmatrix} -2 \\ 3 \end{pmatrix}$. ♢

It is known that SVP is no harder than CVP, since there is a reduction from SVP to CVP. Both are considered to be hard problems and CVP is *NP-hard*.

The classical *Minkowski Theorem* (see [**HPS08**]) gives an upper bound to the norm of the shortest nonzero vector:

Theorem 14.14. *Let Λ be a lattice and $S \subset \mathbb{R}^n$ a convex centrally symmetric set. If the volume of S is greater than $2^n \det(\Lambda)$, then S contains a nonzero lattice point.* ♢

A set S is *centrally symmetric* if $x \in S$ implies $-x \in S$. S is called *convex* if $x, y \in S$ implies $x + t(y - x) \in S$ for $t \in [0, 1]$, i.e., if the line segment between two points $x, y \in S$ is contained in S. For example, balls or cubes with center 0 are centrally symmetric and convex.

Corollary 14.15. $\lambda_1(\Lambda) \leq \sqrt{n}\,(\det(\Lambda))^{\frac{1}{n}}$.

Proof. Let S be a ball with center 0 and radius $\sqrt{n}\,(\det(\Lambda))^{\frac{1}{n}}$. Then

$$v = (\det(\Lambda))^{\frac{1}{n}}(1, 1, \dots, 1) \in S$$

since $\|v\| = (\det(\Lambda))^{\frac{1}{n}}\sqrt{n}$. Furthermore, any vector $w = (\det(L))^{\frac{1}{n}} \cdot (x_1, x_2, \dots, x_n)$ with $|x_i| \leq 1$ for all $i = 1, \dots, n$ lies in S. Hence

$$\left[-(\det(\Lambda))^{\frac{1}{n}}, (\det(\Lambda))^{\frac{1}{n}}\right]^n \subset S.$$

The volume of S is thus greater than $2^n \det(\Lambda)$, the volume of the above n-dimensional cube. S satisfies the prerequisite of Minkowski's Theorem 14.14 and must contain a nonzero lattice vector. $\square$

Remark 14.16. The above upper bound for $\lambda_1(\Lambda)$ can be improved to

$$\sqrt{\gamma_n}\,(\det(\Lambda))^{\frac{1}{n}}$$

using *Hermite's* constant γ_n. For a given dimension n, the constant γ_n is the smallest number such that *every lattice* of rank n contains a nonzero vector v with

$$\|v\| \leq \sqrt{\gamma_n}\,(\det(\Lambda))^{\frac{1}{n}}.$$

For example, $\gamma_2 = \frac{2}{\sqrt{3}}$, but the exact value of γ_n is known only for a few values of n. The expected length of the shortest vector of a *random lattice* is much smaller. Heuristically, the approximate length is the radius of an n-dimensional ball with volume $\det(\Lambda)$. Stirling's asymptotic formula for the volume of an n-dimensional ball of radius r is

$$V_n(r) \approx \frac{1}{\sqrt{n\pi}}\left(\sqrt{\frac{2\pi e}{n}}\,r\right)^n.$$

Now assume that the covolume of a lattice with $\lambda_1(\Lambda) = r$ is approximately the volume of a ball of radius r. Rearranging the above equation gives for large values of n:

$$r \approx \sqrt{\frac{n}{2\pi e}}\,(\det(\Lambda))^{\frac{1}{n}}.$$

This is the *Gaussian heuristic* for randomly chosen lattices of dimension n. $\diamondsuit$

A first approach to SVP is to enumerate all nonzero points $v \in \mathbb{Z}^n$ with $\|v\| \leq C$ (using the bounds above) and to verify whether $v \in \Lambda$. If B is any basis of Λ, then v is a lattice point if and only if $v \in B^{-1}v \in \mathbb{Z}^n$. This algorithm has exponential running time in n.

A similar enumeration method (with exponential running time) can be used to solve CVP: simply test vectors with integer coordinates 'near' the target vector.

Another approach to CVP is *Babai's rounding method.* The rational coordinates of a given target vector w with respect to B are $B^{-1}w$. Rounding the coordinates yields the lattice vector $v = B\lfloor B^{-1}w\rceil$. However, it is not guaranteed that v is the closest lattice vector to w.

Example 14.17. Consider Example 14.3 and let $v = \begin{pmatrix}2\\6\end{pmatrix}$ be a target vector. Looking at Figure 14.1 or by simply trying out vectors near v, it is easy to see that the closest lattice vector is $w = \begin{pmatrix}2\\7\end{pmatrix}$. However, this is only feasible in small dimensions.

We consider B_1 and B_2 (see Example 14.3). For the 'good' basis B_1 we obtain

$$B_1^{-1}v = \frac{1}{10}\begin{pmatrix} 2 & -2 \\ 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 6 \end{pmatrix} = \begin{pmatrix} -\frac{4}{5} \\ \frac{13}{5} \end{pmatrix}.$$

The rounded coordinates are $(-1, 3)$ and the resulting lattice vector is

$$B_1 \begin{pmatrix} -1 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \end{pmatrix},$$

which is in fact the closest vector. Now we use the 'bad' basis B_2:

$$B_2^{-1}v = \frac{1}{10}\begin{pmatrix} 10 & 10 \\ -7 & -4 \end{pmatrix} \begin{pmatrix} 2 \\ 6 \end{pmatrix} = \begin{pmatrix} 8 \\ -\frac{31}{5} \end{pmatrix}.$$

The rounded coordinates are $(8, -6)$ and the corresponding lattice vector is

$$B_2 \begin{pmatrix} 8 \\ -6 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix},$$

and this is only the second-best solution. $\diamondsuit$

Finding short vectors in random q-ary lattices is assumed to be intractable for large dimensions, say several hundreds. Below, we will see that public-key cryptosystems can be based on lattices, where a 'good' basis with short vectors forms the private key and only a 'bad' basis is public.

14.2. Lattice Algorithms

A major task in lattice computations is to give a basis that is as good as possible, i.e., the basis vectors should be short and almost orthogonal. The *LLL algorithm* provides a partial solution: it runs in polynomial time and improves the basis with regard to the above objectives. However, for large dimensions, the LLL basis is far from being optimal and the security of lattice-based cryptography relies on the relative ineffectiveness of the LLL algorithm.

A much simpler task is to transform a set of generators into a basis in standard *echelon form*. This also allows us to decide whether two bases (or two sets of generators) give the same lattice.

Below, we define the *column-style Hermite Normal Form* (or *echelon* form) of an $n \times m$ matrix A of rank n over $\mathbb{Z}$.

Definition 14.18. Let H be an $n \times m$ matrix over $\mathbb{Z}$ such that $rk(H) = n$. We say H is in *Hermite Normal Form* (HNF), if the following conditions are satisfied:

(1) The rightmost $m - n$ columns of H are zero.

(2) H is in lower-triangular form.

(3) All coefficients of H are non-negative.

(4) In each row of H, the unique maximum coefficient lies on the diagonal. $\diamondsuit$

Hence, a matrix in Hermite Normal Form is of the following type:

$$H = \begin{pmatrix} >0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \geq 0 & >0 & \dots & 0 & 0 & \dots & 0 \\ & & \vdots & & & & \\ \geq 0 & \geq 0 & \dots & >0 & 0 & \dots & 0 \end{pmatrix}.$$

HNFs also exist for general $n \times m$ matrices, i.e., without the condition $rk(A) = n$. Since we only consider lattices of full rank, this is not needed here.

Every integer matrix can be transformed into a matrix in HNF form:

Proposition 14.19. *Let A be a $n \times m$ matrix over $\mathbb{Z}$ with $rk(A) = n$ and $n \leq m$. Then there exists a unimodular $m \times m$ matrix U, such that $H = AU$ is in HNF. Furthermore, the columns of A and H generate the same lattice $\Lambda \subset \mathbb{R}^n$.*

Proof. The matrix H can be computed by Gaussian elimination. The following operations are sufficient: swapping two columns, multiplying a column by -1 and adding an integer multiple of a column to another column. These operations are given by a multiplication with a unimodular matrix on the right. We leave the details to the reader. $\square$

Remark 14.20. Above, we considered the *column-style* HNF. There is also a *row-style* HNF H of matrix A in upper-triangular form where $H = UA$. Both HNFs are transposes of each other.

Example 14.21. (1) We compute the HNF of B_1 and B_2 in Example 14.3. Let $B_1 = \{v_1, v_2\} = \left\{ \begin{pmatrix} 4 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix} \right\}$. Set $v_1 \leftarrow (-v_1) + 2v_2$ and swap v_1 and v_2. We obtain the HNF

$$H = \begin{pmatrix} 2 & 0 \\ 2 & 5 \end{pmatrix}.$$

Now consider the second basis $B_2 = \left\{ \begin{pmatrix} 8 \\ -7 \end{pmatrix}, \begin{pmatrix} 10 \\ -10 \end{pmatrix} \right\}$. Set $v_1 \leftarrow (-v_1) + v_2$, giving $\begin{pmatrix} 2 & 10 \\ -3 & -10 \end{pmatrix}$. Now set $v_2 \leftarrow (-5v_1) + v_2$ and $v_1 \leftarrow v_2 + v_1$. This gives $H = \begin{pmatrix} 2 & 0 \\ 2 & 5 \end{pmatrix}$ as above. In fact, B_1 and B_2 generate the same lattice.

(2) Consider a slightly more complicated example:

$$A = \begin{pmatrix} 2 & -6 & 12 & 4 \\ 10 & -30 & 11 & 6 \\ 2 & -6 & 4 & 5 \end{pmatrix}.$$

We let SageMath compute the HNF:

```
sage: A=matrix(ZZ,[[2,-6,12,4],[10,-30,11,6],[2,-6,4,5]])
sage: H=A.transpose().hermite_form().transpose(); H
[ 2  0  0  0]
[ 3  7  0  0]
[14 11 23  0]
```

Since SageMath computes a row-style HNF, we have to add `transpose()` operations. The columns of A and H generate the same lattice; the three nonzero columns of H form a basis of the lattice. ♢

Although many lattices do not possess an orthogonal basis, a good basis should have *almost orthogonal* vectors.

Definition 14.22. The *orthogonality defect* of a basis $b_1, \dots, b_n$ of a lattice Λ is defined by

$$\frac{||b_1|| \cdots ||b_n||}{\det(\Lambda)}.$$ ♢

The orthogonality defect is always ≥ 1. It is close to 1 for a 'good' basis and equal to 1 for an orthogonal basis.

Example 14.23. The orthogonality defect of the 'good' basis B_1 (see Example 14.3) is 1.17 and that of the 'bad' basis B_2 is 15.03. ♢

The main objective of lattice reduction is efficient transformation of an arbitrary basis into an almost orthogonal basis.

For inner product spaces, for example $\mathbb{R}^n$, the *Gram-Schmidt Orthogonalization* (GSO) produces an orthogonal basis (see Algorithm 14.1). If $b_1, \dots, b_n$ is any given basis, then one obtains an orthogonal basis by iterative projections: let $b_1^* = b_1$ and

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \text{ , where } \mu_{i,j} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*} \text{ for } 1 \leq j < i \leq n.$$

The numbers $\mu_{i,j}$ are called GSO coefficients. The summand $\mu_{i,j} b_j^*$ gives the projection of b_i onto b_j^* and their sum is the projection of b_i onto the hyperplane $\langle b_1^*, \dots, b_{i-1}^* \rangle$. The difference of b_i and the projection gives the vector b_i^*, which is orthogonal to all vectors $b_1^*, \dots, b_{i-1}^*$. The vector b_i^* is also called the *projection onto the orthogonal complement* of $b_1^*, \dots, b_{i-1}^*$.

b_i^* can also be computed by successive projections of b_i onto the orthogonal complement of b_j^*, where j runs from $i-1$ to 1. Initially set $b_1^* = b_1, \dots, b_n^* = b_n$ and update each vector $b_2^*, \dots, b_n^*$ recursively:

$$b_i^* \leftarrow b_i^* - \mu_{i,j} b_j^* \text{ , where } j = i-1, \dots, 1.$$

We write B_i for the square norm $||b_i^*||^2$ of vectors in the GSO basis.

Algorithm 14.1 Gram-Schmidt Orthogonalization Algorithm (GSO)

```
Input: Basis $b_1, \dots, b_n$ of $\mathbb{R}^n$
Output: Orthogonal basis $b_i^*, \dots, b_n^*$, GSO coefficients $\mu_{i,j}$
Initialisation: $b_1^* = b_1$
1: for $i = 2$ to $n$ do
2:     $b_i^* = b_i$
3:     for $j = i - 1$ downto 1 do
4:         $\mu_{i,j} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$ // GSO coefficients
5:         $b_i^* \leftarrow b_i^* - \mu_{i,j} b_j^*$
6:     end for
7: end for
8: return $b_i^*, \mu_{i,j}$
```

The standard GSO algorithm needs to be modified for lattices, since the GSO basis is not contained in the lattice unless all GSO coefficients are integers. Now, the obvious approach is to *round* the GSO coefficients $\mu_{i,j}$. Let $\lfloor \mu_{i,j} \rceil$ be the closest integer to $\mu_{i,j}$. Then set

$$b_i = b_i - \lfloor \mu_{i,j} \rceil b_j \text{ for } j = i - 1, \dots, 1.$$

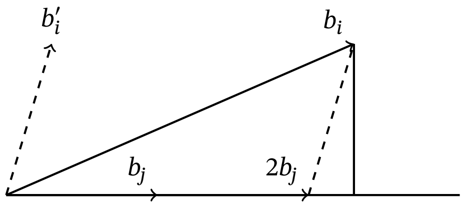


Figure 14.2. Projection of b_i onto the orthogonal complement of b_j and lifting it back to a lattice vector b_i' (dashed). In this example, one has $\lfloor \mu_{i,j} \rceil = 2$. The new basis b_i', b_j is size-reduced.

This can be interpreted as projecting b_i onto the orthogonal complement of b_j and lifting it back to a lattice vector (see Figure 14.2). Since we updated b_i, we also need to update the GSO coefficients $\mu_{i,1}, \dots, \mu_{i,j}$ after each projection. The new GSO coefficient $\mu_{i,k}$ is

$$\frac{b_i \cdot b_k^*}{b_k^* \cdot b_k^*} = \mu_{ik} - \lfloor \mu_{i,j} \rceil \frac{b_j \cdot b_k^*}{b_k^* \cdot b_k^*} = \mu_{i,k} - \lfloor \mu_{i,j} \rceil \mu_{j,k} \text{ for } k = 1, \dots, j.$$

We set $\mu_{j,j} = 1$ and replace μ_{ik} with $\mu_{ik} - \lfloor \mu_{ij} \rceil \mu_{jk}$ for $k = 1, \dots, j$. The last iteration ($k = j$) updates μ_{ij} by $\mu_{ij} - \lfloor \mu_{ij} \rceil$. This is the error when replacing the GSO coefficient μ_{ij} with the integer coefficient $\lfloor \mu_{ij} \rceil$.

Note that b_i is not changed if $\lfloor \mu_{i,j} \rceil = 0$.

Definition 14.24. An ordered basis $b_1, \dots, b_n$ of a lattice Λ is called *size-reduced* if all GSO coefficients $\mu_{i,j}$ satisfy $|\mu_{i,j}| \leq \frac{1}{2}$. ◊

A size-reduced basis cannot be further reduced, since all rounded GSO coefficients are zero. But note that this property depends on the order of the vectors. Even in dimension 2, it may happen that b_1, b_2 is size-reduced while b_2, b_1 is not.

A size-reduced basis can be computed with the following integer variant of the GSO algorithm (see Algorithm 14.2).

Algorithm 14.2 Size Reduction Algorithm

```
Input: Basis $b_1, \dots, b_n$ of lattice $\Lambda$
Output: Reduced basis $b_1, \dots, b_n$, GSO coefficients $\mu_{i,j}$
Initialisation: Run GSO Algorithm 14.1, get GSO basis $b_i^*$ and coefficients $\mu_{i,j}$.
1: for $i = 2$ to $n$ do
2:   for $j = i - 1$ downto 1 do
3:     $b_i \leftarrow b_i - \lfloor \mu_{i,j} \rceil b_j$
4:     for $k = 1$ to $j$ do
5:       $\mu_{i,k} \leftarrow \mu_{i,k} - \lfloor \mu_{i,j} \rceil \mu_{j,k}$ // update GSO coefficients where $\mu_{j,j} = 1$
6:     end for
7:   end for
8: end for
9: return $b_i, b_i^*, \mu_{i,j}$
```

We note that the size reduction algorithm does not change the GSO Basis $b_1^*, \dots, b_n^*$ and their square norms $B_1, \dots, B_n$. A size-reduced basis $b_1, \dots, b_n$ may be further improved by changing the *order* of the vectors. Consider the GSO Algorithm 14.1: if b_i and b_{i+1} are swapped, then the reduction algorithm leaves $b_1^*, \dots, b_{i-1}^*$ and $b_{i+2}^*, \dots, b_n^*$ unchanged and the new value for b_i^* is

$$b_{i+1} - \sum_{j=1}^{i-1} \mu_{i+1,j} b_j^*.$$

In terms of GSO vectors without swapping, this is the same as

$$b_{i+1}^* + \mu_{i+1,i} b_i^*.$$

We compare $||b_i^*||^2 = B_i$ (i.e., without swapping) with

$$||b_{i+1}^* + \mu_{i+1,i} b_i^*||^2 = B_{i+1} + \mu_{i+1,i}^2 B_i$$

(with swapping). If $B_{i+1} + \mu_{i+1,i}^2 B_i < B_i$ then swapping reduces the norm of b_i^* and should therefore be applied to the lattice basis. In this case, one swaps b_i and b_{i+1} and again applies the size-reduction algorithm.

Definition 14.25. Let $b_1, \dots, b_n, b_1^*, \dots, b_n^*, B_1, \dots, B_n$ be as above and $\delta \in]\frac{1}{4}, 1[$. Then the *Lovacz condition* with factor δ is defined by

$$\delta B_i \leq B_{i+1} + \mu_{i+1,i}^2 B_i$$

for $i = 1, \dots, n-1$. The condition is equivalent to $(\delta - \mu_{i+1,i}^2)B_i \leq B_{i+1}$. An ordered basis $b_1, \dots, b_n$ is called δ-LLL-reduced if it is size-reduced and the Lovacz condition holds with factor δ. ♢

A typical choice is $\delta = \frac{3}{4}$, which ensures that the algorithm terminates in polynomial time (in contrast to $\delta = 1$). Now we can give the basic version of the famous LLL (Lenstra-Lenstra-Lovasz) algorithm (see Algorithm 14.3).

Algorithm 14.3 LLL Lattice Reduction Algorithm

```
Input: Basis $b_1, \dots, b_n$ of lattice $\Lambda$
Output: LLL-reduced basis $b_1, \dots, b_n$
1: Run Algorithm 14.2, get size-reduced basis $b_1, \dots, b_n$, GSO basis $b_i^*$
   and GSO coefficients $\mu_{i,j}$.
2: Compute $B_i = ||b_i^*||^2$ for $1 \leq i \leq n$
3: for $i = 1$ to $n-1$ do
4:    if $(\delta - \mu_{i+1,i}^2)B_i > B_{i+1}$ then
5:       Swap $b_i$ and $b_{i+1}$
6:       Goto Step 1
7:    end if
8: end for
9: return $b_1, \dots, b_n$
```

Remark 14.26. Algorithm 14.3 can be optimized: it is not necessary to leave the loop and to re-run the size-reduction Algorithm 14.2, if b_i and b_{i+1} are swapped (step 5). Instead, it is sufficient to update the GSO basis and several GSO coefficients and to decrease the loop index i by 1.

Example 14.27. We consider the following HNF basis of a 3-dimensional lattice (see Example 14.21 (2)):

$$b_1 = \begin{pmatrix} 2 \\ 3 \\ 14 \end{pmatrix}, \; b_2 = \begin{pmatrix} 0 \\ 7 \\ 11 \end{pmatrix}, \; b_3 = \begin{pmatrix} 0 \\ 0 \\ 23 \end{pmatrix}.$$

We apply the LLL lattice reduction Algorithm 14.3. First, we compute the GSO coefficients:

$$\mu_{21} = \frac{b_2 \cdot b_1}{b_1 \cdot b_1} = \frac{175}{209}, \; \mu_{31} = \frac{b_3 \cdot b_1}{b_1 \cdot b_1} = \frac{322}{209}, \; \mu_{32} = \frac{b_3 \cdot b_2^*}{b_2^* \cdot b_2^*} = -\frac{3473}{4905}.$$

Note that μ_{32} is computed using the updated vector

$$b_2^* = b_2 - \mu_{21} b_1 = (-350/209, 938/209, -151/209)^T.$$

The size-reduction algorithm sets $b_2 = b_2 - \lfloor \mu_{21} \rceil b_1 = b_2 - b_1 = (-2, 4, -3)^T$. Then, μ_{21} is updated to $\mu_{21} - \lfloor \mu_{21} \rceil = -\frac{34}{209}$.

Now, $b_3 = b_3 - \lfloor \mu_{32} \rceil b_2 = b_3 + b_2 = (-2, 4, 20)^T$. The GSO coefficient μ_{31} changes to $\mu_{31} - \lfloor \mu_{32} \rceil \mu_{21} = \mu_{31} + \mu_{21} = \frac{288}{209}$ and μ_{32} is updated to $\mu_{32} - \lfloor \mu_{32} \rceil = \mu_{32} + 1 = \frac{1432}{4905}$.

The last size-reduction step is $b_3 = b_3 - \lfloor \mu_{31} \rceil b_1 = b_3 - b_1 = (-4, 1, 6)^T$. The GSO coefficient μ_{31} changes to $\mu_{31} - \lfloor \mu_{31} \rceil = \mu_{31} - 1 = \frac{79}{209}$. We have thus computed a new size-reduced basis:

$$b_1 = \begin{pmatrix} 2 \\ 3 \\ 14 \end{pmatrix}, \; b_2 = \begin{pmatrix} -2 \\ 4 \\ -3 \end{pmatrix}, \; b_3 = \begin{pmatrix} -4 \\ 1 \\ 6 \end{pmatrix}.$$

We check the Lovacz condition for $\delta = \frac{3}{4}$ and $i = 1$. We have $B_1 = 209$, $B_2 = \frac{4905}{209}$ and

$$(\delta - \mu_{21}^2) B_1 \approx 151 > B_2 \approx 23.$$

The Lovacz condition is not satisfied, and so we swap b_1 and b_2:

$$b_1 = \begin{pmatrix} -2 \\ 4 \\ -3 \end{pmatrix}, \; b_2 = \begin{pmatrix} 2 \\ 3 \\ 14 \end{pmatrix}, \; b_3 = \begin{pmatrix} -4 \\ 1 \\ 6 \end{pmatrix}.$$

Apply the GSO algorithm and obtain $B_1 = 29$, $B_2 = \frac{4905}{29}$ and $B_3 = \frac{103684}{4905}$ for the new basis. The GSO coefficients are:

$$\mu_{21} = -\frac{34}{29}, \; \mu_{31} = -\frac{6}{29}, \; \mu_{32} = \frac{2087}{4905}.$$

Then size reduction is applied and we update b_2 by $b_2 - \lfloor \mu_{21} \rceil b_1 = b_2 + b_1 = (0, 7, 11)^T$ and μ_{21} by $\mu_{21} + 1 = -\frac{6}{29}$. The other two coefficients μ_{31} and μ_{32} are not changed and are rounded to 0. Hence the new basis

$$b_1 = \begin{pmatrix} -2 \\ 4 \\ 3 \end{pmatrix}, \; b_2 = \begin{pmatrix} 0 \\ 7 \\ 11 \end{pmatrix}, \; b_3 = \begin{pmatrix} -4 \\ 1 \\ 6 \end{pmatrix}$$

is size-reduced. The Lovacz condition is satisfied for $i = 1$:

$$(\delta - \mu_{21}^2) B_1 \approx 21 < B_2 \approx 169,$$

but is not satisfied for $i = 2$:

$$(\delta - \mu_{32}^2) B_2 \approx 96 > B_3 \approx 21.$$

We have to swap b_2 and b_3 and obtain

$$b_1 = \begin{pmatrix} -2 \\ 4 \\ -3 \end{pmatrix}, \; b_2 = \begin{pmatrix} -4 \\ 1 \\ 6 \end{pmatrix}, \; b_3 = \begin{pmatrix} 0 \\ 7 \\ 11 \end{pmatrix}.$$

Now the GSO algorithm gives $B_1 = 29$, $B_2 = \frac{1501}{29}$ and $B_3 = \frac{103684}{1501}$, and the updated GSO coefficients are

$$\mu_{21} = -\frac{6}{29},\ \mu_{31} = -\frac{5}{29},\ \mu_{32} = \frac{2087}{1501}.$$

Again, the size-reduction algorithm is applied. b_2 does not change since $\lfloor \mu_{21} \rceil = 0$. We update b_3 by $b_3 - \lfloor \mu_{32} \rceil b_2 = b_3 - b_2 = (4,6,5)^T$, μ_{31} by $\mu_{31} - \lfloor \mu_{32} \rceil \mu_{21} = \frac{1}{29}$ and μ_{32} by $\mu_{32} - 1 = \frac{586}{1501}$. Now the basis b_1, b_2, b_3 is size-reduced and both Lovacz conditions are satisfied:

$$(\delta - \mu_{21}^2)B_1 \approx 21 < B_2 \approx 52,$$
$$(\delta - \mu_{32}^2)B_2 \approx 31 < B_3 \approx 69.$$

The LLL algorithm stops and outputs the LLL-reduced basis

$$b_1 = \begin{pmatrix} -2 \\ 4 \\ -3 \end{pmatrix},\ b_2 = \begin{pmatrix} -4 \\ 1 \\ 6 \end{pmatrix},\ b_3 = \begin{pmatrix} 4 \\ 6 \\ 5 \end{pmatrix}.$$

We verify our result with SageMath:

```
sage: A=matrix([[2,0,0],[3,7,0],[14,11,23]])
sage: A.transpose().LLL().transpose()
[-2 -4  4]
[ 4  1  6]
[-3  6  5]
```

The LLL-reduced basis is shorter than the original basis and one can show (for example, by testing shorter vectors with integer coefficients) that $b_1 = \begin{pmatrix} -2 \\ 4 \\ -3 \end{pmatrix}$ is the shortest vector of the lattice Λ. $\diamondsuit$

One can show that the LLL algorithm always terminates and runs in polynomial time. The number of swaps and hence the number of executions of the main loop is bounded by $O(n^2 \ln(X))$, where X is an upper bound on the norms of the input vectors. We refer to [**Gal12**] and [**HPS08**] for a proof the following statement:

Theorem 14.28. *Let Λ be a lattice in $\mathbb{Z}^n$ with basis $b_1, b_2, \dots, b_n$ and $\|b_i\| \leq X$ for $i = 1, \dots, n$. Let $\frac{1}{4} < \delta < 1$; then the LLL algorithm terminates and the running time is $O(n^6 \ln(X)^3)$.* $\diamondsuit$

The next Proposition relates the norms of the LLL-reduced basis to the norms of the GSO basis.

Proposition 14.29. *Let $b_1, \dots, b_n$ be an LLL-reduced basis with $\delta = \frac{3}{4}$. Let $b_1^*, \dots, b_n^*$ be the corresponding GSO basis and $B_i = \|b_i^*\|^2$ as above. Then:*

(1) *$B_i \leq 2B_{i+1}$ for $1 \leq i < n$ and $B_j \leq 2^{i-j}B_i$ for $1 \leq j \leq i \leq n$.*

(2) *$B_i \leq \|b_i\|^2 \leq (\frac{1}{2} + 2^{i-2})B_i$ for $1 \leq i \leq n$.*

(3) *$\|b_j\| \leq 2^{(i-1)/2}\|b_i^*\|$ for $1 \leq j \leq i \leq n$.*

(4) *$\lambda_1(\Lambda) \geq \min_{1\leq i\leq n} \|b_i^*\|$.*

Proof. Since the basis is reduced, one has $\mu_{i+1,i}^2 \leq \frac{1}{4}$. The Lovacz condition for $\delta = \frac{3}{4}$ implies (1). The GSO construction gives

$$b_i = b_i^* + \sum_{j=1}^{i-1} \mu_{i,j} b_j^*.$$

Since the GSO vectors are orthogonal, one obtains $\|b_i^*\| \leq \|b_i\|$ and

$$\|b_i\|^2 = B_i + \sum_{j=1}^{i-1} \mu_{i,j}^2 B_j.$$

Furthermore, $\mu_{i,j}^2 B_j \leq \frac{1}{4}B_j \leq \frac{1}{4}2^{(i-j)}B_i$ by (1). This gives part (2), since

$$\|b_i\|^2 \leq B_i\left(1 + \frac{1}{4}\sum_{j=1}^{i-1} 2^{i-j}\right) = B_i\left(1 + \frac{1}{4}(2^i - 2)\right) = B_i\left(\frac{1}{2} + 2^{i-2}\right).$$

For $j \geq 1$ we have $\frac{1}{2} + 2^{j-2} \leq 2^{j-1}$. Thus (2) implies $\|b_j\|^2 \leq 2^{j-1}B_j$. Since $B_j \leq 2^{i-j}B_i$ by (1), we obtain $\|b_j\|^2 \leq 2^{j-1}2^{i-j}B_i = 2^{i-1}B_i$. Taking square roots proves part (3).

Suppose v is a shortest nonzero lattice vector and $v = \sum_{i=1}^n x_i b_i$ where $x_i \in \mathbb{Z}$; then:

$$v = \sum_{i=1}^n \left(x_i b_i^* + \sum_{j=1}^{i-1} x_i \mu_{i,j} b_j^* \right) = \sum_{i=1}^n (x_i + \mu_{i+1,i}x_{i+1} + \cdots + \mu_{n,i}x_n)\, b_i^*.$$

Now, let i be the largest index such that $x_i \neq 0$. The above formula and the orthogonality of the GSO basis implies $\|v\| \geq |x_i|\,\|b_i^*\|$ and hence part (4). □

The next Proposition shows how effective the LLL algorithm is (in the worst case) with respect to computing a short vector and an almost orthogonal basis. The algorithm is good for small values of n, but the bounding factors grow exponentially in n.

Proposition 14.30. *Let $b_1, \dots, b_n$ be an LLL-reduced basis with $\delta = \frac{3}{4}$. Then:*

(1) $\|b_1\| \leq 2^{(n-1)/2}\lambda_1(\Lambda)$.

(2) $\det(\Lambda) \leq \prod_{i=1}^n \|b_i\| \leq 2^{n(n-1)/4}\det(\Lambda)$.

(3) $\|b_1\| \leq 2^{(n-1)/4}\det(\Lambda)^{\frac{1}{n}}$.

Proof. We use the previous Proposition 14.29. Part (1) gives $\|b_i^*\| \geq 2^{(1-i)/2}\|b_1^*\|$. Part (4) and $b_1 = b_1^*$ yield the first inequality:

$$\lambda_1(\Lambda) \geq \min_{1\leq i\leq n} \|b_i^*\| \geq \min_{1\leq i\leq n} 2^{(1-i)/2}\|b_1^*\| = 2^{(1-n)/2}\|b_1\|.$$

We have $\det(\Lambda) = \prod_{i=1}^{n} \|b_i^*\|$. Inequality (2) follows from $\|b_i^*\| \leq \|b_i\|$ and part (3) of Proposition 14.29:

$$\|b_i^*\| \leq \|b_i\| \leq 2^{(i-1)/2}\|b_i^*\|.$$

Furthermore, $\|b_1\| \leq 2^{(i-1)/2}\|b_1^*\|$ gives

$$\|b_1\|^n \leq \prod_{i=1}^{n} 2^{(i-1)/2}\|b_i^*\| = 2^{n(n-1)/4}\det(\Lambda),$$

which yields part (3). □

14.3. GGH Cryptosystem

In 1997, Goldreich, Goldwasser and Halevi introduced a public-key encryption and a signature scheme based on lattices (GGH, [**GGH97**]). The security relies on the hardness of the closest vector problem (CVP). Although the original scheme has flaws, and it was shown that GGH is much easier than the general closest vector problem (CVP), the underlying construction is still relevant today. Together with the Atjai-Dwork and NTRU schemes, GGH is one of the classical lattice-based cryptosystems. Furthermore, the GGH scheme is very efficient.

Definition 14.31. Let $n \in \mathbb{N}$ be the security parameter. Choose $M \in \mathbb{N}$ and a small number σ, for example $M = n$ and $\sigma = 3$. The GGH encryption scheme is defined as follows:

- The plaintext space is $\mathcal{M} = \{-M, \dots, 0, 1, \dots, M\}^n$ and the ciphertext space is $\mathcal{C} = \mathbb{Z}^n$.
- For key generation, one chooses a uniform $n \times n$ matrix B over $\mathbb{Z}$ with small entries, say integers between -4 and 4. Verify that B is invertible (over $\mathbb{Q}$), and otherwise generate a new matrix. B is a ‘good’ basis of a lattice $\Lambda \subset \mathbb{R}^n$ (since the basis has small coefficients) and forms the *private key*. Let H be the HNF of B. This is a ‘bad’ basis of Λ and defines the *public key*. Alternatively, H could be any random basis of Λ. However, H can be efficiently reduced to the HNF anyway.
- A plaintext $m \in \{-M, \dots, 0, 1, \dots, M\}^n$ is encrypted by choosing a uniform random noise vector $r \in \{-\sigma, \sigma\}^n$ and computing the ciphertext
$$c = Hm + r.$$
- For decryption, one uses Babai’s rounding method to recover the plaintext
$$m = H^{-1}B\lfloor B^{-1}c\rceil.$$
◊

The ciphertext is close to the lattice point Hm and the assumption is that finding this vector given c is hard. The following Proposition shows that decryption is correct if the noise vector r is small.

Proposition 14.32. *Let B, H, Λ and r be as above. If $\lfloor B^{-1}r\rceil = 0$ then GGH decryption is correct.*

Proof. Let $c = Hm + r$. Since $Hm \in \Lambda$, we have $B^{-1}Hm \in \mathbb{Z}^n$ and

$$\lfloor B^{-1}c\rceil = \lfloor B^{-1}Hm + B^{-1}r\rceil = B^{-1}Hm + \lfloor B^{-1}r\rceil.$$

It follows that

$$H^{-1}B\lfloor B^{-1}c\rceil = H^{-1}BB^{-1}Hm + H^{-1}B\lfloor B^{-1}r\rceil = m + H^{-1}B\lfloor B^{-1}r\rceil.$$

The second summand is zero if $\lfloor B^{-1}r\rceil = 0$. This proves the assertion. □

Example 14.33. Let $n = 4$, $M = 4$ and $\sigma = 1$. We choose a short basis B of a lattice Λ and compute its HNF:

$$B = \begin{pmatrix} 2 & -3 & 1 & -4 \\ -2 & 1 & 0 & 4 \\ -1 & 3 & 2 & 1 \\ -1 & -4 & 3 & -2 \end{pmatrix}, \quad H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 44 & 18 & 4 & 49 \end{pmatrix}.$$

Suppose the plaintext is $m = (3, -4, 1, 3)$ and the noise vector $r = (-1, 1, 1, -1)$ is chosen. The associated ciphertext is:

$$c = Hm + r = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 44 & 18 & 4 & 49 \end{pmatrix}\begin{pmatrix} 3 \\ -4 \\ 1 \\ 3 \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 3 \\ -4 \\ 1 \\ 211 \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ -3 \\ 2 \\ 210 \end{pmatrix}.$$

Decryption uses the private basis B. We compute:

$$B^{-1} = \frac{1}{49}\begin{pmatrix} 61 & 45 & 10 & -27 \\ -10 & -13 & 8 & -2 \\ 29 & 23 & 16 & -4 \\ 33 & 38 & 3 & -13 \end{pmatrix}, \quad B^{-1}c = \frac{1}{7}\begin{pmatrix} -809 \\ -55 \\ -117 \\ -396 \end{pmatrix}, \quad \lfloor B^{-1}c\rceil = \begin{pmatrix} -116 \\ -8 \\ -17 \\ -57 \end{pmatrix}.$$

We recover the plaintext:

$$H^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{44}{49} & -\frac{18}{49} & -\frac{4}{49} & \frac{1}{49} \end{pmatrix},$$

$$m = H^{-1}B\lfloor B^{-1}c\rceil = H^{-1}B\begin{pmatrix} -116 \\ -8 \\ -17 \\ -57 \end{pmatrix} = H^{-1}\begin{pmatrix} 3 \\ -3 \\ 1 \\ 211 \end{pmatrix} = \begin{pmatrix} 3 \\ -4 \\ 1 \\ 3 \end{pmatrix}.$$

However, if an adversary tries to decrypt c using the public basis H, then the result m' differs from m:

$$m' = \lfloor H^{-1}c \rceil = \left\lfloor \begin{pmatrix} 2 \\ -3 \\ 2 \\ \frac{24}{7} \end{pmatrix} \right\rceil = \begin{pmatrix} 2 \\ -3 \\ 2 \\ 3 \end{pmatrix}. \qquad \diamondsuit$$

It can be shown that the GGH encryption scheme has inherent weaknesses. A major problem is that the noise vector r has to be short for correct decryption.

As a consequence, GGH ciphertexts are not uniformly distributed and can be distinguished from random data, but then the closest vector problem is much easier than in the general case. Practical attacks could be mounted for $n < 400$, and larger dimensions are impractical because of the key size that grows quadratically in n. There are proposals for improvements of GGH that require further cryptanalysis.

14.4. NTRU

The NTRU cryptosystem was invented in the 1990s by Hoffstein, Pipher and Silverman [**HPS98**]. The classical definition of NTRU uses polynomials in the ring

$$R = \mathbb{Z}[x]/(x^N - 1),$$

where N is fixed, for example $N = 743$. Furthermore, a large modulus q and a small modulus p are needed with $\gcd(p, q) = 1$, for example $q = 2048$ and $p = 3$. We begin with the classical definition of NTRU and outline the relation to lattices later in this section.

Multiplication in the ring R can be viewed as a *convolution product*, since

$$(a_0 + a_1x + \cdots + a_{N-1}x^{N-1})(b_0 + b_1x + \cdots + b_{N-1}x^{N-1}) \equiv c_0 + \cdots + c_{N-1}x^{N-1},$$

where $c_k = \sum_{i+j \equiv k \bmod N} a_i b_j$. The convolution product is often denoted by a '$*$', but since it is also the usual multiplication in the quotient ring R (see Proposition 4.60) we will not use a special notation.

NTRU uses polynomials with small coefficients. We define $\mathcal{T}(d_1, d_2) \subset R$ to be the subset of *ternary polynomials*, where $f \in \mathcal{T}(d_1, d_2)$ if a representative f of degree $< N$ has d_1 coefficients equal to 1, d_2 coefficients equal to -1 and the remaining coefficients equal to zero (see [**HPS08**]).

Definition 14.34. Let N, p be prime numbers and q, $d \in \mathbb{N}$ such that $\gcd(p, q) = \gcd(N, q) = 1$ and $q > (6d + 1)p$. Let $R = \mathbb{Z}[x]/(x^N - 1)$, $R_p = \mathbb{Z}_p[x]/(x^N - 1)$ and $R_q = \mathbb{Z}_q[x]/(x^N - 1)$. Then the NTRU cryptosystem is defined as follows:

- The plaintext space is $\mathcal{M} = R_p$ and the ciphertext space is $\mathcal{C} = R_q$.

- For key generation, two polynomials $f, g \in R$ are chosen such that $f \in \mathcal{T}(d+1,\, d)$ and $g \in \mathcal{T}(d,\, d)$. Moreover, f must be invertible $\bmod q$ and $\bmod p$, i.e., there exist polynomials $f_p, f_q \in R$ such that
$$f \cdot f_p \equiv 1 \bmod p \;\text{ and }\; f \cdot f_q \equiv 1 \bmod q.$$
Then set
$$h = f_q \cdot g \bmod q.$$
The public key is $pk = (N, p, q, h)$ and the private key is $sk = f$.
- For encryption, the plaintext is encoded into a polynomial m of degree less than N with integer coefficients between $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, i.e., a lifted element of the ring R_p. A random polynomial $r \in R$ is chosen such that $r \in \mathcal{T}(d,\, d)$. The ciphertext is defined by
$$c = p\,rh + m \bmod q.$$
- For decryption, compute
$$a = fc \bmod q$$
and represent a with integer coefficients between $-\frac{q}{2}$ and $\frac{q}{2}$. The plaintext is recovered by
$$m = f_p\, a \bmod p. \qquad \diamondsuit$$

We verify that decryption is correct:
$$\begin{aligned} a = fc &= f(prh + m) \bmod q \\ &= pfrf_q\, g + fm \bmod q \\ &= p\,r\,g + fm \bmod q. \end{aligned}$$
The above restrictions on the coefficients r, g, f and m ensure that the magnitude of the largest coefficient of $p\,r\,g + fm$ is at most
$$p \cdot 2d + (2d+1)\,\frac{p}{2} = \left(3d + \frac{1}{2}\right) p.$$
Now the assumption $q > (6d+1)p$ implies that every coefficient of $p\,r\,g + fm$ has magnitude less than $\frac{q}{2}$. Therefore, the equation $a = p\,r\,g + fm$ holds over the integers before reducing modulo q. This gives
$$f_p\, a = pf_p\, r\, g + f_p\, fm \equiv 0 + m = m \bmod p.$$

Note that the plain NTRU cryptosystem is not CPA-secure, since it leaks a small part of the plaintext (see Exercise 10).

Remark 14.35. There are recommendations for N, p, q and the distribution of coefficients in the polynomials f, g and r can be more general than explained above. It is estimated that NTRU encryption with $N = 743$, $p = 3$, $q = 2^{11} = 2048$ achieves a very high level of security [**HPS⁺17**].

Example 14.36. We look at a toy example. Let $N = 5$, $p = 3$, $q = 29$, $d = 1$. For key generation, we choose

$$f = x^4 + x^3 - 1 \in \mathcal{T}(2,1) \text{ and } g = x^3 - x^2 \in \mathcal{T}(1,1).$$

We use SageMath to check that f is invertible modulo p and q in $R = \mathbb{Z}[x]/(x^5 - 1)$.

```
sage: p=3; q=29
sage: mod=x^5 -1
sage: R=PolynomialRing(ZZ,x).quotient_ring(mod)
sage: Rp=PolynomialRing(Integers(p),x).quotient_ring(mod)
sage: Rq=PolynomialRing(Integers(q),x).quotient_ring(mod)
sage: f = x^4 + x^3 - 1; g = x^3 - x^2
sage: fp=1/Rp(f);fq=1/Rq(f);fp;fq
2*xbar^3 + 2*xbar^2 + xbar + 2
24*xbar^4 + 8*xbar^3 + 3*xbar^2 + 11*xbar + 13
```

Hence $f_p = -x^3 - x^2 + x - 1$ and $f_q = -5x^4 + 8x^3 + 3x^2 + 11x + 13$. We obtain

$$\begin{aligned} h = f_q\, g &= (-5x^4 + 8x^3 + 3x^2 + 11x + 13)(x^3 - x^2) \\ &= -5x^7 + 8x^6 + 3x^5 + 11x^4 + 13x^3 + 5x^6 - 8x^5 - 3x^4 - 11x^3 - 13x^2 \\ &= -5x^7 + 13x^6 - 5x^5 + 8x^4 + 2x^3 - 13x^2 \\ &= 8x^4 + 2x^3 + 11x^2 + 13x - 5 \mod 29. \end{aligned}$$

The last equation is true since $x^5 = 1$ in R. We have generated the public key $pk = (N, p, q, h)$, and the private key is $sk = f$.

Suppose the plaintext is encoded in the polynomial $m = x^3 + x$. A random polynomial $r = x^4 - x \in \mathcal{T}(1,1)$ is chosen for encryption and the ciphertext is

$$\begin{aligned} c = p\,rh + m &= 3(x^4 - x)(8x^4 + 2x^3 + 11x^2 + 13x - 5) + x^3 + x \\ &= 8x^4 + 21x^3 + 25x^2 + 20x + 15 \mod 29. \end{aligned}$$

```
sage: h=fq*Rq(g)
sage: m=x^3+x; r = x^4-x
sage: c= Rq(p)*Rq(h)*Rq(r)+Rq(m);c
8*xbar^4 + 21*xbar^3 + 25*xbar^2 + 20*xbar + 15
```

Note that $m(1) = c(1) \mod 29$.

In order to decrypt c, we use the secret key f and compute

$$a = fc \mod q = (x^4 + x^3 - 1)(8x^4 + 21x^3 + 25x^2 + 20x + 15) \mod 29.$$

SageMath gives the following result:

```
sage: a=Rq(f)*c;a
27*xbar^4 + 2*xbar^3 + 4*xbar^2 + 26*xbar + 1
```

Hence $a = -2x^4 + 2x^3 + 4x^2 - 3x + 1$. We verify that $a = prg + fm$ holds in R, without reducing modulo q. Finally, $f_p\, a \mod p$ recovers the plaintext $m = x^3 + x$.

```
sage: a=p*R(r)*R(g)+R(f)*R(m); a
-2*xbar^4 + 2*xbar^3 + 4*xbar^2 - 3*xbar + 1
sage: Rp(fp)*Rp(a)
xbar^3 + xbar
```

$\Diamond$

Now we describe the *lattice representation* of NTRU. It is easy to see that elements in $R = \mathbb{Z}[x]/(X^N - 1)$ correspond to vectors in $\mathbb{Z}^N$: a polynomial $f = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1} \in R$ is mapped to the vector $\widetilde{f} = (a_0, a_1, \ldots, a_{N-1}) \in \mathbb{Z}^N$ of coefficients. This is clearly a group isomorphism, but how does the multiplication in R translate to $\mathbb{Z}^N$? We define the *circulant* matrix of f as

$$C_f = \begin{pmatrix} a_0 & a_1 & \ldots & a_{N-1} \\ a_{N-1} & a_0 & \ldots & a_{N-2} \\ & \ldots & & \\ a_1 & a_2 & \ldots & a_0 \end{pmatrix}.$$

Let $g = b_0 + b_1 x + \cdots + b_{N-1}x^{N-1}$. Then $(b_0\ b_1\ \ldots\ b_{N-1})\, C_f$ gives the coefficients of the convolution product $fg = gf \in R$, i.e.,

$$(b_0\ b_1\ \ldots\ b_{N-1})\, C_f = \tilde{g}\, C_f = \widetilde{fg}.$$

Suppose an NTRU cryptosystem as above is given; then

$$fh = f f_q\, g = g \mod q.$$

Hence the coefficient vectors satisfy $\widetilde{f} C_h = \tilde{g} \mod q$. Now we define the associated *NTRU lattice* Λ of dimension $2N$. The lattice is generated by the *rows* of the matrix

$$A = \begin{pmatrix} I_N & C_h \\ 0 & qI_N \end{pmatrix},$$

where I_N is the $N \times N$ identity matrix. The matrix A is derived from the *public key* of the NTRU cryptosystem. Since $g = fh + qu$ for some polynomial $u \in R$, we have

$$(\widetilde{f},\ \tilde{u})A = \left((\widetilde{f},\ \tilde{u}) \begin{pmatrix} I_N \\ 0 \end{pmatrix}, (\widetilde{f},\ \tilde{u}) \begin{pmatrix} C_h \\ qI_N \end{pmatrix} \right) = (\widetilde{f},\ \tilde{g}),$$

and hence $(\widetilde{f},\ \tilde{g}) \in \Lambda$. Note that f forms the private key. Since f and g have small coefficients, $(\widetilde{f},\ \tilde{g})$ is a short vector in Λ. NTRU can therefore be attacked by finding short vectors in the lattice Λ. However, this is assumed to be intractable if the dimension is large enough.

Example 14.37. We continue Example 14.36. The corresponding NTRU lattice Λ of dimension 10 is generated by the rows of the following matrix:

$$A = \begin{pmatrix} I_5 & C_h \\ 0 & 29I_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -5 & 13 & 11 & 2 & 8 \\ 0 & 1 & 0 & 0 & 0 & 8 & -5 & 13 & 11 & 2 \\ 0 & 0 & 1 & 0 & 0 & 2 & 8 & -5 & 13 & 11 \\ 0 & 0 & 0 & 1 & 0 & 11 & 2 & 8 & -5 & 13 \\ 0 & 0 & 0 & 0 & 1 & 13 & 11 & 2 & 8 & -5 \\ 0 & 0 & 0 & 0 & 0 & 29 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 29 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 29 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 29 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 29 \end{pmatrix}.$$

The lattice is given by a ‘bad’ basis of long vectors. We want to attack this NTRU cryptosystem by finding a ‘good’ basis and the short secret vector $(\widetilde{f},\ \tilde{g}) \in \Lambda$. We use SageMath to compute the LLL-reduced basis of the lattice (see Section 14.2):

$$\begin{pmatrix} 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 \\ -1 & -1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 1 & -1 & 0 & 0 \\ -4 & -4 & 5 & 0 & 5 & 5 & 5 & 5 & -5 & -10 \\ 0 & -5 & 9 & -5 & 0 & 10 & 0 & -5 & 0 & -5 \\ 0 & -5 & 5 & 1 & 0 & 10 & 9 & 5 & 5 & 0 \\ -5 & 9 & -5 & 0 & 0 & 0 & -5 & 0 & -5 & 10 \\ 0 & 5 & -5 & -5 & 6 & -5 & -9 & 4 & 5 & 5 \end{pmatrix}.$$

Since the dimension is small, our attack is successful: the negative of the first row is

$$(\widetilde{f}, \tilde{g}) = (-1, 0, 0, 1, 1,\ 0, 0, -1, 1, 0).$$

Thus we could derive the private key $f = -1 + x^3 + x^4$ as well as $g = -x^2 + x^3$ from the public key (N, p, q, h).

Remark 14.38. The security of NTRU relies on the assumption that, given $h = f^{-1}g \bmod q$, it is hard to recover f and g. There are many potential attacks and NTRU has been investigated for two decades. It is generally assumed that improved NTRU schemes are secure and can even achieve IND-CCA2 security if the parameter recommendations are observed. Some doubts remain with respect to the cyclotomic structure of the ring $R = \mathbb{Z}[x]/(x^N - 1)$ and there are proposals [**BCLvV18**] replacing $x^N - 1$ with $x^N - x - 1$ and choosing a prime q instead of a power of 2 (compare Remark 14.35).

14.5. Learning with Errors

Learning with Errors (LWE) is a mathematical problem with many applications in post-quantum cryptography. LWE was introduced by O. Regev [**Reg09**] and is based on a system of *noisy* (approximate) linear equations modulo an integer q. Suppose the row vectors $a_i \in \mathbb{Z}_q^n$ are chosen uniformly at random, $s \in \mathbb{Z}_q^n$ is a secret vector and a linear system of $m \geq n$ approximate equations is given:

$$\begin{aligned} a_1 \cdot s &\approx b_1 \mod q, \\ a_2 \cdot s &\approx b_2 \mod q, \\ &\vdots \\ a_m \cdot s &\approx b_m \mod q. \end{aligned}$$

The errors $e_1, e_2, \dots, e_m$ of each equation are small and are drawn from a distribution χ on $\mathbb{Z}$. The distribution is usually taken to be a discrete Gaussian (see Definition 14.39 below). The number of equations is typically chosen to be large enough that the approximate linear system of equations has a unique solution with high probability. The *LWE search problem* is to find (to *learn*) the unknown vector $s \in \mathbb{Z}_q^n$ given a_i and b_i for $i = 1, \dots, m$. If all errors are zero, then the problem can be solved in polynomial time by Gaussian elimination. However, this does not work for approximate equations because even a small error vector can change the solution completely.

The distribution χ of error terms is quite important in LWE. If the error is zero or too small then LWE is easy. On the other hand, if the error terms are large or even uniform in $\mathbb{Z}_q$, then the LWE problem has more than one solution. It is also important that the error is randomly chosen, because otherwise LWE is easy. A standard choice of the error distribution χ is the *Discrete Gaussian*.

Definition 14.39. A *discrete Gaussian* is a probability distribution on $\mathbb{Z}$ obtained from a continuous Gaussian. The probability mass function of a discrete Gaussian is proportional to a continuous Gaussian with mean 0 and standard deviation σ *at each integer value* (see Figure 14.3), i.e.,

$$p(x) = \frac{1}{c}\, e^{-\frac{x^2}{2\sigma^2}} \text{ for } x \in \mathbb{Z},$$

where $c = \sum_{k\in\mathbb{Z}} e^{-k^2/(2\sigma^2)} \approx \sigma\sqrt{2\pi}$. A discrete Gaussian is said to have *parameter* or *width* s, if $\sigma = \frac{s}{\sqrt{2\pi}}$, and a discrete Gaussian distribution of width s is denoted by $D_{\mathbb{Z},s}$. The probability mass function of $D_{\mathbb{Z},s}$ is proportional to $e^{-\pi x^2/s^2}$. ♢

Definition 14.40. Let $n, m, q \in \mathbb{N}$, $m \geq n$, $q \geq 2$ and assume that the secret vector $s \in \mathbb{Z}_q^n$ and the $m \times n$ matrix A over $\mathbb{Z}_q$ are chosen uniformly. Let χ be a probability distribution on $\mathbb{Z}$ and $e \in \mathbb{Z}_q^m$ a vector that is drawn from the distribution χ^m.

(1) The *Search-LWE* problem is to find s given A and $b = As + e$.

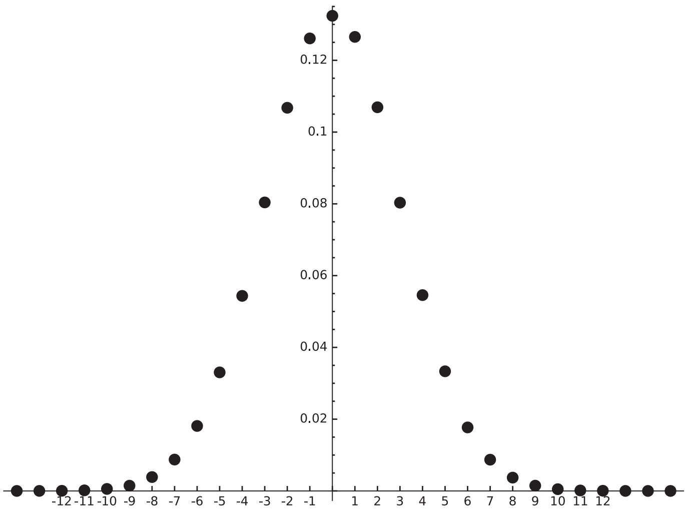


Figure 14.3. Probability mass function of the discrete Gaussian distribution $D_{\mathbb{Z},s}$ with standard deviation $\sigma = 3$ and width $s = \sqrt{2\pi}\sigma \approx 7.5$.

(2) The *Decision-LWE* problem is to distinguish with a non-negligible advantage between the cases i) $b = As+e$, and ii) $b \in \mathbb{Z}_q^m$ is chosen uniformly at random, given A and b.

The LWE problem depends on the parameters n, m, q and χ. ◊

It is easy to see that solving the Search-LWE problem also solves the decision problem, and it can be shown that the decision and the search version of LWE are equivalent if q is bounded by a polynomial in n (see [**Reg09**]).

LWE can be viewed as a lattice problem: the matrix A defines a q-ary lattice $\Lambda_q(A^T)$ of dimension m. The search problem is to find the closest lattice vector v to a given noisy vector $v + e$ where e is chosen according to χ. This is also called a *Bounded Distance Decoding* (BDD) problem. The decision problem is to distinguish between a uniform random vector b and a noisy lattice vector $v + e$.

The following theorem ([**Reg09**]) is one of the key results and explains why the LWE problem is believed to be hard:

Theorem 14.41. *Let $n \in \mathbb{N}$ be the security parameter, let m, $q \in \mathbb{N}$ be polynomial in n and let $\chi = D_{\mathbb{Z},s}$ be a discrete Gaussian of parameter s such that $s = \alpha q > 2\sqrt{n}$ and $0 < \alpha < 1$. Then solving the LWE decision problem is at least as hard as quantumly solving SIVP$_\gamma$ on arbitrary n-dimensional lattices, where $\gamma = \tilde{O}(n/\alpha)$.* ◊

In other words: if a polynomial-time algorithm achieves a non-negligible advantage in the LWE decision problem (in the average case), then a polynomial-time algorithm on a quantum computer can solve any instance (also the worst case) of the

$\mathrm{SIVP}_{n/\alpha}$ problem. Since the approximation of the shortest independent vector problem to within polynomial factors is assumed to be a hard problem even for quantum computers, the LWE problem is probably hard. The *worst-case to average-case* reduction of Theorem 14.41 is particularly interesting, because most other cryptographic constructions are based on *average-case* hardness.

Now we define a public-key cryptosystem that is based on LWE and has the same strong security guarantee.

Definition 14.42. Let $n,\ m,\ q \in \mathbb{N}, m \geq n, q \geq 2$ and χ an error distribution on $\mathbb{Z}$. Let l be the plaintext length. Then the LWE public-key cryptosystem is defined by:

- The plaintext space $\mathcal{M} = \{0,1\}^l \cong \mathbb{Z}_2^l$.
- The ciphertext space $\mathcal{C} = \mathbb{Z}_q^n \times \mathbb{Z}_q^l$.
- For key generation, one chooses an $n \times l$ matrix S and an $m \times n$ matrix A uniformly at random and an $m \times l$ matrix E according to χ. All matrices are defined over $\mathbb{Z}_q$. The private key is $sk = S$ and the public key is $pk = (A, P)$, where $P = AS + E$.
- To encrypt a plaintext $v \in \{0,1\}^l$, one chooses a vector $a \in \{0,1\}^m$ uniformly at random. The ciphertext is given by
$$(u,c) = \mathcal{E}_{pk}(v) = \left(A^T a, P^T a + \lfloor \tfrac{q}{2} \rceil v\right) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l.$$
- Decryption of a ciphertext (u,c) is defined by
$$\mathcal{D}_{sk}(u,c) = \left\lfloor \lfloor \tfrac{q}{2} \rceil^{-1} (c - S^T u) \right\rceil \bmod 2.$$
In other words, each coordinate of $c - S^T u$ is tested whether it is closer to 0 or to $\lfloor \frac{q}{2} \rceil$ modulo q. ♢

The decryption may have errors. One has
$$\begin{aligned}\mathcal{D}_{sk}(\mathcal{E}_{pk}(v)) &= \mathcal{D}_{sk}(A^T a, P^T a + \lfloor \tfrac{q}{2} \rceil v)\\ &= \left\lfloor \lfloor \tfrac{q}{2} \rceil^{-1} ((AS+E)^T a + \lfloor \tfrac{q}{2} \rceil v - S^T A^T a) \right\rceil \bmod 2\\ &= \left\lfloor \lfloor \tfrac{q}{2} \rceil^{-1} (E^T a + \lfloor \tfrac{q}{2} \rceil v) \right\rceil \bmod 2\\ &= \left\lfloor \lfloor \tfrac{q}{2} \rceil^{-1} E^T a \right\rceil + v \bmod 2.\end{aligned}$$

We can assume that the coordinates of $E^T a$ are integers between $-\frac{q}{2}$ and $\frac{q}{2}$. Then an decryption error occurs if the magnitude of a coordinate of $E^T a$ is greater than or equal to $\frac{q}{4}$. Each coordinate is a sum of at most m coefficients of E^T, and E is chosen according to χ. Suppose that χ is a discrete Gaussian $D_{\mathbb{Z},s}$ of width s. Then the coordinates of the vector $E^T a$ have magnitude less than $\sqrt{m}s$ with high probability. One can show that for all n there are choices for q, m and s such that the error probability is very small and the LWE problem is hard (see [**MR09**] and [**Pei14**]).

The security of LWE encryption relies on the hardness of the LWE problem (see [**Reg09**]):

Theorem 14.43. *If the LWE decision problem with parameters n, m, q and χ is hard, then the LWE encryption scheme has indistinguishable encryption under chosen plaintext attack (IND-CPA secure).* ◊

A disadvantage of the above LWE cryptosystem is a substantial encryption blowup factor, since l plaintext bits are transformed into $(n+l)$ size(q) ciphertext bits. The public key (A, P) has $(mn+ml)$ size(q) bits, and the size of the secret key S is nl size(q) bits.

Example 14.44. We illustrate the LWE encryption scheme with a toy example. A more realistic example can be found in Exercise 13. Let $n = 4$, $q = 23$, $m = 8$, $\alpha = \frac{5}{23}$ and a discrete Gaussian distribution $D_{\mathbb{Z},s}$ of width $s = 5$ and standard deviation $\sigma = \frac{s}{\sqrt{2\pi}} \approx 2$. We choose uniform random matrices S (secret) and A (public):

$$A = \begin{pmatrix} 9 & 5 & 11 & 13 \\ 13 & 6 & 6 & 2 \\ 6 & 21 & 17 & 18 \\ 22 & 19 & 20 & 8 \\ 2 & 17 & 10 & 21 \\ 10 & 8 & 17 & 11 \\ 5 & 16 & 12 & 2 \\ 5 & 7 & 11 & 7 \end{pmatrix} \quad S = \begin{pmatrix} 5 & 2 & 9 & 1 \\ 6 & 8 & 19 & 1 \\ 19 & 18 & 9 & 18 \\ 9 & 2 & 14 & 18 \end{pmatrix}.$$

The secret matrix E is chosen according to $D_{\mathbb{Z},s}$. The matrix $P = AS + E$ is public.

$$E = \begin{pmatrix} 0 & 22 & 1 & 21 \\ 0 & 22 & 22 & 22 \\ 22 & 22 & 22 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 \\ 1 & 22 & 1 & 22 \\ 22 & 0 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 10 & 5 & 21 & 7 \\ 3 & 1 & 13 & 1 \\ 19 & 15 & 6 & 13 \\ 9 & 20 & 0 & 16 \\ 8 & 17 & 13 & 4 \\ 15 & 21 & 20 & 17 \\ 0 & 12 & 3 & 19 \\ 16 & 2 & 7 & 15 \end{pmatrix}.$$

We want to encrypt $v = (1, 0, 1, 1)^T$ and choose a random vector $a \in \{0, 1\}^8$:

$$a = (1, 1, 0, 1, 0, 0, 0, 1)^T.$$

We have $\lfloor \frac{23}{2} v \rceil = (12, 0, 12, 12)^T$ and compute the ciphertext

$$(u, c) = \mathcal{E}_{pk}(v) = (A^T a, P^T a + \lfloor \tfrac{23}{2} \rceil v) = \big((3, 14, 2, 7)^T, (4, 5, 7, 5)^T\big) \bmod 23.$$

For decryption, we use the secret matrix S and obtain

$$c - S^T u = (4, 5, 7, 5)^T - S^T (3, 14, 2, 7)^T = (11, 21, 12, 10)^T \bmod 23.$$

Coefficients close to 0 mod 23 give the bit 0 and coefficients close to $\lfloor \frac{23}{2} \rceil = 12$ give the bit 1. Hence we recover the plaintext $v = (1, 0, 1, 1)^T$. ♢

LWE encryption can be attacked by recovering the key S. Let s be a column of S and e, b the corresponding columns of E and P, respectively. Then

$$As + e = b$$

and the LWE problem is to find s (or As) given A and b. This is a closest vector problem (CVP) for the lattice $\Lambda_q(A^T)$ and the target vector b. One can use *Kannan's embedding technique* and convert CVP into a shortest vector problem (SVP). Let H be an $m \times m$ matrix over $\mathbb{Z}$ such that the columns form a basis of $\Lambda_q(A^T)$. We construct a lattice Λ' of dimension $m + 1$ that is generated by the columns of the $(m + 1) \times (m + 1)$ matrix

$$B = \begin{pmatrix} H & b \\ 0 & M \end{pmatrix},$$

where $M > 0$ is an embedding factor. Now the equation $-As + b = e \mod q$ implies that a linear combination of the columns of B gives the vector $e' = \begin{pmatrix} e \\ M \end{pmatrix} \in \Lambda'$. It is likely that e' is the shortest vector in the lattice Λ' generated by B, if the parameter M is properly chosen. Thus, we will probably find e by solving the SVP problem for the lattice Λ' of dimension $m+1$. However, if the shortest vector in $\Lambda_q(A^T)$ is smaller than e, then the method does not work.

Example 14.45. We continue Example 14.44 above and try to attack the LWE problem. Our aim is to first find column s of the private matrix S given only A and P. First we compute the HNF of the 23-ary lattice Λ of dimension 8 generated by A.

```
sage: AZ=A.lift().augment(23*identity_matrix(8))
      H=AZ.transpose().hermite_form().transpose()
```

The lattice Λ is generated by the columns of A (where the coefficients are lifted to $\mathbb{Z}$) and $23\mathbb{Z}^8$. The first eight (nonzero) columns of the HNF H form a basis of Λ. Now we construct the 9-dimensional lattice Λ':

```
sage: B=H[:,0:8].augment(P[:,0]).stack(vector([0,0,0,0,0,0,0,0,2]))
```

We choose the embedding factor $M = 2$, but the reader may check that $M = 1$ and $M = 3$ also work in this example. Finally, we compute the LLL-reduced basis.

```
sage: B.transpose().LLL().transpose()
[ 0  1 -2  0  2 -4 -2  4 -1]
[ 0  0 -2 -2 -3  0  0  1 -3]
[-1  0  0 -1  0 -3 -3 -1  1]
[ 0 -2 -1 -1  1  0 -1  2  0]
[ 0  1 -3 -2  0  0  0  1  3]
[ 1 -3 -1  1  0 -2  2 -2 -2]
[ 1  1  0 -4  2  0  2  0 -3]
```

```
[-1 -1  0 -1  1 -1  3  1  0]
[ 2  0  2  0 -2  0  0  2  2]
```

Since the dimension is small, the first column $(0,0,-1,0,0,1,1,-1,2)^T$ is the shortest nonzero vector of Λ'. We have successfully found the vector $\begin{pmatrix} e \\ 2 \end{pmatrix}$, where $e \bmod 23$ is the secret error vector (the first column of E; see Example 14.44). ♢

A disadvantage of LWE is that the key size and the number of operations is at least quadratic in the main security parameter n. Optimizations of Regev's LWE encryption scheme exist, for example the more compact Lindner-Peikert scheme [**LP11**].

The key size and the efficiency of the system was also the motivation behind the development of an algebraic variant of LWE called *Ring-LWE* (R-LWE). We mention R-LWE only briefly and refer to the literature for a detailed discussion and applications to encryption and key exchange (see [**LPR13**] and subsequent works).

LWE is based on the hardness to find a vector s given the matrix A and a noisy product $As+e$ over $\mathbb{Z}_q$. Now, Ring-LWE leverages the *cyclotomic ring*

$$R = \mathbb{Z}_q[x]/(x^n+1),$$

where n is a power of 2. The matrix A and the secret vectors s and e are replaced by elements in R. Since all elements of R can be uniquely represented by polynomials of degree less than n, we have $R \cong \mathbb{Z}_q^n$. Any element $a \in R$ generates a principal ideal $(a) = \{a\,x \mid x \in R\} \subset R$. If $a \neq 0$, then (a) corresponds to an n-dimensional q-ary *ideal lattice*.

The R-LWE problem is to find $s \in R$ given $a \in R$ and $b = as + e \in R$. The ring element e is 'small' and chosen according to an error distribution. Note that as is an element of the ideal lattice and only the noisy element b is given to an adversary. An encryption scheme can be defined in a similar way to Definition 14.42 above. The main advantage is that the key length and the number of operations are now linear in n.

Ring-LWE also has an asymptotic security guarantee: there is a reduction from a *worst-case lattice problem* SVP_γ to R-LWE, i.e., solving R-LWE is at least as hard as quantumly solving the SVP_γ problem on arbitrary ideal lattices. Note that the reduction is based on *ideal lattices* in R instead of general q-ary lattices. Such ideal lattices have additional structure and it might be possible that efficient attacks will be found that exploit the algebraic structure and cannot be applied to general lattices. However, such attacks are not yet known and may not exist.

14.6. Summary

- Lattices are discrete subgroups of $\mathbb{R}^n$. They have a finite basis over $\mathbb{Z}$.
- Finding the shortest vector in a lattice (SVP), the shortest independent vectors (SIVP) and the closest lattice vector to a given vector (CVP) are supposed to be hard problems.
- The LLL-algorithm leverages the Gram-Schmidt Orthogonalization (GSO) algorithm to reduce the size of a given basis.
- A short, almost orthogonal basis of a lattice can form the private key of a public-key encryption scheme.
- GGH and NTRU are classical cryptosystems based on lattice problems, and improved versions of NTRU are assumed to be secure.
- The LWE (Learning with Errors) decision problem is to distinguish between a noisy lattice vector and a uniform random vector. LWE has a security guarantee that is based on the worst-case hardness of an approximate SIVP problem. The corresponding LWE encryption scheme is CPA-secure if the LWE problem is hard.

Exercises

1. Consider Example 14.3. Find a unimodular matrix $\tilde{U}$ such that $B_1 = B_2\tilde{U}$.
2. Consider Example 14.10. Find the lattice $\Lambda_{10}^{\perp}(A)^*$.
3. Use the lattice defined by the columns of $\begin{pmatrix} 1 & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} \end{pmatrix}$ to show that $\gamma_2 \geq \frac{2}{\sqrt{3}}$.

   Remark: In fact, equality holds.
4. Consider the lattice in Example 14.3. Use Babai's rounding method to find the closest lattice vector to the target vector $v = (42, 25)$. Can you also use the basis B_2?
5. Let A be an $n \times m$ matrix over $\mathbb{Z}_q$. Show that $\Lambda_q^{\perp}(A) = q\Lambda_q(A)^*$ and $\Lambda_q(A) = q\Lambda_q^{\perp}(A)^*$.
6. Let Λ be a 100-dimensional lattice of covolume 2^{104}. Give an upper bound for $\lambda_1(\Lambda)$ and the expected length of the shortest vector if Λ is randomly chosen.
7. Let Λ be a lattice with the following HNF basis:

$$b_1 = \begin{pmatrix} -13 \\ 31 \end{pmatrix}, \; b_2 = \begin{pmatrix} 0 \\ 47 \end{pmatrix}.$$

   (a) Compute the determinant of Λ and the orthogonality defect of the basis $\{b_1, b_2\}$.

(b) Apply the LLL-algorithm: First, run the GSO and the size reduction algorithm.
(c) Check the Lovacz condition. Swap the basis vectors and again apply the size reduction algorithm.
(d) Check the Lovacz condition again and output the LLL-reduced basis. Compute the orthogonality defect of this basis.
(e) Give the shortest nonzero vector of Λ.

8. A lattice Λ is generated by the columns of the following matrix:

$$H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 14 & 18 & 63 \end{pmatrix}.$$

(a) Encrypt $m = (-2, -3, 1)$ with the GGH encryption scheme. Choose the noise vector $r = (1, -1, -1)$.
(b) Try to decrypt the ciphertext c using the matrix H. Why does this attempt fail?
(c) Apply the LLL-algorithm to H using SageMath and show that Λ has the following short basis:

$$B = \begin{pmatrix} -2 & -1 & 4 \\ -2 & 1 & -3 \\ -1 & 4 & 2 \end{pmatrix}.$$

(d) Decrypt c using the private basis B.

9. Use the NTRU cryptosystem of Example 14.36 to encrypt $m = -x^4 - x^2 + 1$. Choose the random polynomial $r = -x^3 + x^2$, then decrypt the ciphertext and recover m.

10. Show that the NTRU cryptosystem is not CPA-secure. How could you fix this problem? *Hint:* The construction of r implies $r(1) = 0 \bmod q$ and hence $c(1) = m(1) \bmod q$ (compare Example 14.36).

11. Use the key pair of the LWE Example 14.44.
(a) Choose a random vector a and encrypt $v = (1, 0, 0, 1)^T$.
(b) Decrypt the ciphertext (u, c).
(c) Can decryption errors occur with the chosen matrix E? Given an example of E and a such that the decryption is not correct.

12. Consider Example 14.45. Use Kannan's embedding technique to recover the second column of the secret error matrix E. Show that the embedding factors $M = 1,\ 2,\ 3$ work. What happens if you set $M = 4$?

13. Construct an example of LWE encryption with the parameters $n = l = 136$, $q = 2003$, $m = 2008$, $\alpha = 0.0065$, $s = \alpha q$ and a discrete Gaussian distribution $D_{\mathbb{Z},s}$ with $\sigma = \frac{s}{\sqrt{2\pi}} \approx 5.19$ (see [**MR09**] Table 3).

(a) Generate a key pair:

```
sage: Zq=IntegerModRing(2003)
sage: A=random_matrix(Zq,2008,136)
sage: S=random_matrix(Zq,136,136)
```

Make sure to write the `import` statement into one line.

```
sage: from sage.stats.distributions.discrete_gaussian_integer import
      DiscreteGaussianDistributionIntegerSampler
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=5.19)
```

```
sage: numbers=[D() for _in range(2008*136)]
sage: E = Matrix(Zq, 2008, numbers)
sage: P=A*S+E
```

(b) We choose a random plaintext v of length $l = 136$ and a random vector a of length $m = 2008$.

```
sage: v=[ZZ.random_element(0,2) for _in range(136)]
sage: v=vector(v)
sage: a=[ZZ.random_element(0,2) for _in range(2008)]
sage: a=vector(a)
```

(c) Compute the ciphertext (u, c). Note that $\lfloor \frac{q}{2} \rceil = 1002$.

```
sage: u=A.transpose()*a
sage: c=P.transpose()*a+1002*v
```

(d) We need a map that takes an integer modulo q as input, multiplies it by $\lfloor \frac{q}{2} \rceil^{-1}$, rounds to the nearest integer and outputs the result modulo 2.

```
sage: def transform(x):
            y=RR(x.lift())*1/1002
            return ZZ(round(y))
```

(e) Decrypt the ciphertext:

```
sage: w=vector(map(transform, c-S.transpose()*u))
```

(f) Compare the result with the original plaintext, for example by printing out $v - w$.

(g) Print out $1002v$ and $c - S^T u$ and interpret the result.

(h) Explain why or why not any decryption errors occurred.

(i) Give the sizes of the public key, the private key, the plaintext and the ciphertext.

Chapter 15

Code-based Cryptography

Error correction codes play an important role when data is sent over noisy channels, for example over wireless links, or stored on potentially unreliable media. Channel coding deals with random errors and not with manipulations by adversaries, but integrity protection is a common objective of channel coding and cryptography. However, codes also aim to ensure *error correction*, which goes beyond error detection.

A channel encoder takes an information word as input and generates a *codeword* that is transmitted over a channel. The ratio between the lengths of the original data word and the codeword determines the *information rate* of the code. Decoding is a potentially complex task, where received words are transformed into codewords and the original information is (hopefully) recovered. Codes with good error-correction capabilities, a high information rate and efficient decoding algorithms are available and widely used in practice.

For cryptographic applications, one can use very long codes with a secret structure. In this case, decoding should be hard for an adversary without access to the hidden structure.

In Section 15.1, we give a short introduction to linear codes. Bounds on the parameters of codes are given in Section 15.2. In Section 15.3, we explain classical Goppa codes. The McEliece cryptosystem is based on Goppa codes and represents one of the promising candidates for post-quantum cryptography. We explore the McEliece scheme and the related Niederreiter cryptosystem in Section 15.4.

There are a number of similarities between lattice-based and code-based cryptography. Lattices and codes are linear subspaces of high-dimensional spaces, and for a given target vector, finding the closest vector in the subspace can be a hard problem.

Both use a secret structure which allows for an efficient solution to the problem. However, lattices and codes use a different metric.

Two recommended textbooks on coding theory are [**Rot06**] and [**Bla03**]. A short introduction to error correcting codes and the McEliece cryptosystem is given in [**TW06**]. More details on code-based cryptography can be found in [**OS09**].

15.1. Linear Codes

The basic idea of channel coding is to send (or store) encoded data instead of the original information. The data is replaced by a sequence of codewords and sent over a noisy communication channel or stored on unreliable media (see Figure 15.1). We will only consider *error-correcting codes* and focus on *block codes*, where each information word is separately encoded into a codeword. We note that other error-correcting codes exist, for example convolutional and Turbo codes. There are also other types of coding, in particular source coding.

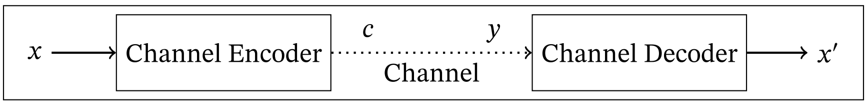


Figure 15.1. Channel coding.

Example 15.1. The most elementary example is *repetition codes*, where an information symbol (for example one bit) is repeated n times, say $n = 3$:

$$s_1 s_2 \cdots \longrightarrow s_1 s_1 s_1 \; s_2 s_2 s_2 \dots .$$

Error detection is straightforward: if the message received does not have this pattern, an error has occurred. The error detection does not always work since a codeword could accidentally change into another codeword. Although the probability of this happening is small, it is not impossible (unless n is very large, which makes the code impractical). Note the difference to hash values or message authentication codes which (almost) always detect changes.

The repetition code also allows for error correction within certain limits: by choosing the symbol with the highest frequency (maximum likelihood) in a received block, up to $\lfloor \frac{n-1}{2} \rfloor$ errors can be corrected. For example, one error can be corrected with $n = 3$, two errors with $n = 5$, etc.

A major disadvantage of a repetition code is the message extension by a factor of n. ♢

Below, we assume that messages and codewords are vectors over a finite field of characteristic 2, i.e., over a binary field $GF(q)$, where $q = 2$ or $q = 2^m$. Coding theory can be studied over arbitrary finite fields, but in practice fields of characteristic 2 are the most important case.

Definition 15.2. A *block code* of length n over $GF(q)$ is a subset C of $GF(q)^n$. The elements of C are called *codewords*. The *code size* is $M = |C|$ and the dimension of C is $k = \log_q(|C|)$. A *linear code* over $GF(q)$ is a linear subspace $C \subset GF(q)^n$. The code size of a linear code is $|C| = q^k$ and C is called a *linear* $[n, k]$ code. The *information rate* of C is given by $R = \frac{k}{n}$.

Example 15.3. The codewords of a repetition code of length n over $GF(q)$ form the one-dimensional subspace $C \subset GF(q)^n$ generated by the vector $(1, 1, \dots, 1)$. C is a linear $[n, 1]$ code over $GF(q)$ and the information rate is $\frac{1}{n}$. ♢

The error-correction capability of a code is determined by the *minimum distance* between the codewords.

Definition 15.4. Let $C \subset GF(q)^n$ be a block code over $GF(q)$.

(1) The Hamming weight $wt(x)$ of $x \in GF(q)^n$ is the number of nonzero coordinates of x.
(2) The Hamming distance $d(x, y) = wt(x - y)$ between two vectors x, $y \in GF(q)^n$ is the number of coordinates where x and y differ.
(3) The minimum distance $d = d(C)$ of a code C is the minimum of all distances $d(c, c')$ for c, $c' \in C$ and $c \neq c'$. A linear $[n, k]$ code with minimum distance d is also called an $[n, k, d]$ code.

Proposition 15.5. *Let C be linear code. The minimum distance $d = d(C)$ is equal to the minimum weight* $\min_{c \in C} wt(c)$.

Proof. Suppose $c, c' \in C$ and $d(c, c') = d$; then $c - c' \in C$ and $wt(c - c') = d$. Conversely, if $c \in C$ and $wt(c) = w$, then $d(c, 0^n) = w$. Note that the zero vector is a codeword since C is a linear code. □

Example 15.6. Consider the repetition code $C = \{0000,\ 1111\}$ of length 4 over $GF(2)$. The minimum distance is $d(C) = 4$ and C is a $[4, 1, 4]$ code. ♢

The following properties of the Hamming distance are easy to show:

Proposition 15.7. *The Hamming distance d is a* metric *on $GF(q)^n$, i.e., for all x, y, $z \in GF(q)^n$ one has:*

(1) $d(x, y) \geq 0$ *with equality if and only if $x = y$.*
(2) $d(x, y) = d(y, x)$ *(symmetry).*
(3) $d(x, z) \leq d(x, y) + d(y, z)$ *(triangle inequality).* ♢

A decoder maps a received word $y \in GF(q)^n$ to a codeword $c \in C$. *Maximum-likelihood decoding* means to decode a word y to the codeword $c \in C$ that maximizes the conditional probability

$$P[\, y \mid c\,].$$

Nearest-codeword decoding of a received word y picks the closest codeword c with respect to the Hamming distance. For a binary symmetric channel with error probability less than one half, maximum-likelihood decoding is equivalent to nearest-codeword decoding. We refer to textbooks on coding theory for a more detailed discussion and now use nearest-codeword decoding.

The following Theorem provides bounds for error detection and correction:

Theorem 15.8. *A code can detect up to $d(C)-1$ errors and correct up to $\lfloor\frac{d(C)-1}{2}\rfloor$ errors.*

Proof. By definition of the minimum distance, adding less than $d(C)$ errors cannot give another codeword and hence up to $d(C) - 1$ errors can be detected. Now let y be a received word having at most $\lfloor\frac{d(C)-1}{2}\rfloor$ errors; then a codeword $c \in C$ exists with $d(y, c) \leq \lfloor\frac{d(C)-1}{2}\rfloor$. Suppose there is another codeword $c' \in C$ such that $d(y, c') \leq \lfloor\frac{d(C)-1}{2}\rfloor$. Then the triangle inequality implies

$$d(c, c') \leq d(c, y) + d(y, c') \leq 2\left\lfloor\frac{d(C)-1}{2}\right\rfloor \leq d(C) - 1,$$

a contradiction. □

Since linear codes are linear subspaces, they can be represented by the span of a set of linearly independent vectors. Writing these vectors into the *rows* of a matrix gives the *generator matrix* G of a code. If C is a linear $[n, k]$ code, then G is a $k \times n$ matrix over $GF(q)$. The set of codewords can be computed by xG, where x runs over all vectors $x \in GF(q)^k$.

The generator matrix of a code is not uniquely determined: adding the multiple of one row to another row or swapping two rows does not change the subspace. Swapping two columns gives an equivalent code, where only the coordinates are permuted. By applying elementary row operations (Gauss-Jordan elimination) and column permutations (if necessary), one can find a generator matrix in systematic form:

$$G = (I_k|\,P) = \left(\begin{array}{cc|c} 1 & 0 & \\ \dots & & P \\ 0 & 1 & \end{array}\right).$$

I_k is the $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix. The corresponding code is called *systematic* and (by Gauss-Jordan elimination) all codes are equivalent to a systematic code. Systematic codes have the advantage that codewords contain the original data as their first k symbols. Otherwise, the information word x must be recovered from a codeword xG by solving a linear system of equations.

How can we verify that a received vector y is a codeword without comparing the vector to a list of all codewords? Using Gaussian elimination, one can check whether $y \in C$, i.e., whether y is a linear combination of the rows of G. A more direct way is using a *parity-check matrix.*

Definition 15.9. Let C be a code with generator matrix G. Then H is called a *parity-check matrix* of C if

$$yH^T = 0 \Longleftrightarrow y \in C.$$

For a received word $y \in GF(q)^n$, the vector yH^T is called the *syndrome* of y.

Proposition 15.10. *Let $G = (I_k|P)$ be the generator matrix of a systematic $[n,k]$ code. Then the $(n-k) \times n$ matrix*

$$H = (-P^T | \, I_{n-k}) = \left(\begin{array}{c|ccc} & 1 & & 0 \\ -P^T & & \dots & \\ & 0 & & 1 \end{array} \right)$$

is the associated parity check matrix.

Proof. Let

$$(v,w) \in GF(q)^k \times GF(q)^{n-k}$$

be a row vector of length n; then

$$(v,w)H^T = \left(H \begin{pmatrix} v^T \\ w^T \end{pmatrix} \right)^T = \left(-P^T v^T + w^T \right)^T = -vP + w.$$

This is the zero vector if and only if $w = vP$, which is equivalent to $(v,w) = vG$ and to (v,w) being a codeword. □

Example 15.11. The $[7,4]$ Hamming code over $GF(2)$ can be defined by the following generator matrix:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The information rate is $\frac{4}{7}$ and the parity-check matrix is

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

We want to show that the minimum distance is $d = 3$. Firstly, the minimum distance cannot be greater than 3, since codewords of weight 3 exist (see Proposition 15.5), for example $c = (1,0,0,0,1,1,0)$.

Now assume that $d(v,w) = 1$ for codewords v and w. Then $wt(v-w) = 1$ and $(v-w)H^T$ is a zero column of H, a contradiction. If $d(v,w) = 2$ then $(v-w)H^T$, a sum of two columns of H, is zero, and hence two columns of H are linearly dependent. However, this is not the case.

The $[7,4,3]$ Hamming code can correct one error. For example, suppose that the vector $y = (1,1,1,0,0,1,1)$ is received. The syndrome is $yH^T = (0,1,1)$ and hence y is not a codeword. However, $c = (1,1,0,0,0,1,1)$ is a codeword because $cH^T = (0,0,0)$. Furthermore, the Hamming distance of y and c is only 1. Therefore, the nearest-codeword decoding of y gives $x = (1,1,0,0)$. ♢

In the above example, one could guess the nearest codeword. A better method is *syndrome decoding*.

Definition 15.12. Let $C \subset GF(q)^n$ be a linear code and $y \in GF(q)^n$ any vector. The set $y + C = \{y + c \mid c \in C\}$ is called a *coset* of C. A vector having minimum Hamming weight in a coset is called a *coset leader*. ♢

Note that all vectors in a coset $y + C$ have the same syndrome yH^T, since $cH^T = 0$ for all codewords $c \in H$ and hence

$$(y + c)H^T = yH^T.$$

Furthermore, all vectors in a given coset can be decoded with the coset leader as their error vector. In fact, the coset leader represents the least change that transforms a vector into a codeword.

Proposition 15.13. *Let C be a linear code and suppose one wants to decode a received word y. Let e be the coset leader of the coset $y + C$, i.e., the coset leader with the syndrome yH^T. Then the nearest codeword to y is $y - e \in C$.*

Proof. Since y and e have the same syndrome, the syndrome of $y-e$ is zero and $y-e$ is a codeword. Furthermore, the coset leader e has minimum weight among the vectors e with $y - e \in C$. □

Example 15.14. We continue Example 15.11. The syndrome is $yH^T = (0,1,1)$ and we need to find the coset leader which has that syndrome. In general, one would use a table that contains the coset leader for each syndrome. In our example, we suspect that y only has a single bit error. Hence the coset leader is a unit vector and its syndrome is a column of the parity-check matrix H. The syndrome $(0,1,1)$ appears in the third column. The coset leader is $e = (0,0,1,0,0,0,0,0)$ and we decode y to the codeword $y - e = (1,1,0,0,0,1,1)$.

Remark 15.15. It is known that the nearest-codeword problem for random codes is hard (NP-complete). This suggests that large codes can be used for cryptographic purposes.

15.2. Bounds on Codes

A good code $C \subset GF(q)^n$ should have a large number of codewords and a large minimum distance, but given n, both parameters cannot be large at the same time. The following Theorem provides an *upper bound* for the number of codewords in terms of the minimum distance.

Theorem 15.16. (*Singleton Bound*) *Let C be a block code of length n, minimum distance d and size M; then*

$$M \leq q^{n-d+1}.$$

In particular, if C is a linear $[n, k, d]$ code, then

$$k \leq n - d + 1 \Longleftrightarrow d \leq n - k + 1.$$

Proof. Consider the map $f : C \rightarrow GF(q)^{n-d+1}$, which removes the first $d-1$ coordinates, i.e., $f(x_1, \ldots, x_n) = (x_d, \ldots, x_n)$. The map is injective, since two codewords differ in at least d coordinates. This shows the first assertion. For a linear code, we have $M = q^k$, and therefore $k \leq n - d + 1$. □

Definition 15.17. A linear $[n, k, d]$ code that satisfies $d = n - k + 1$ is called an *MDS code* (maximum distance separable). ♢

For example, the parity code is MDS (see Exercise 1), but the $[7, 4, 3]$ Hamming code (see Example 15.11) is not MDS. *Reed-Solomon codes* are a well-known class of MDS codes, for which we refer to textbooks on coding theory.

One can use the generator matrix or the parity-check matrix to check whether a code is MDS. We refer to [**Rot06**] for the following fact:

Proposition 15.18. *Let C be a linear $[n, k]$ code with generator matrix G and parity-check matrix H. Then C is MDS if and only if one of the following conditions is satisfied:*

(1) *Every set of $n - k$ columns of H is linearly independent.*

(2) *Every set of k columns of G is linearly independent.*

(3) *If $G = (I_k \mid P)$ is in systematic form, then P and every square submatrix of P is nonsingular.*

Example 15.19. (1) We again consider the $[7, 4]$ Hamming code (see Example 15.11). We see that the first three columns of H are linearly dependent. The last four columns of G are also linearly dependent. One can show that no non-trivial linear code over $GF(2)$ can be MDS.

(2) We construct a systematic $[8, 4]$ code over $GF(2^8)$ with generator matrix $G = (I_4 \mid P)$, where

$$P = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}.$$

P defines the MixColumns operation of the AES block cipher (see Section 5.2). All quadratic submatrices of P are nonsingular over $GF(2^8)$ (see Exercise 6 of Chapter 5), so the code is MDS.

We now show that $k \times k$ matrices P associated to $[2k, k, k+1]$ MDS codes have good *diffusion* properties. The branch number, defined by

$$\min_{v \neq 0}(wt(v^T) + wt(Pv^T)),$$

where v is a row vector of length k, measures the worst-case diffusion of P. The weight of unit vectors is 1 and so the maximum branch number is $k+1$. Let $G = (I_k|P)$ be the generator matrix of a $[2k, k, k+1]$ MDS code and $v \neq 0$. It follows that

$$k+1 \leq wt(vG) = wt(v||vP) = wt(v) + wt(vP) = wt(v^T) + wt(P^T v^T),$$

and so the branch number of P^T is maximal. By Proposition 15.18 (3), the code with generator matrix $(I_k|P^T)$ is also MDS and hence the branch number of P is maximal, too. In this case, P is called *optimal*. The MixColumns matrix of the AES cipher is optimal and the branch number is 5. ♢

There is also a *lower bound* for the size of at least one (not necessarily linear) code with a given length and minimum distance. First, we count the number of vectors in a ball of radius r:

Proposition 15.20. *Let $r \in \mathbb{N}$ and $v \in GF(q)^n$. Then the number of vectors $w \in GF(q)^n$ such that $d(v, w) \leq r$ is*

$$V_q(n, r) = \sum_{i=0}^{r} \binom{n}{i}(q-1)^i.$$

Proof. Let $v \in GF(q)^n$ and $i \leq n$. Then the number of vectors w such that $d(v, w) = i$ is $\binom{n}{i}(q-1)^i$, because there are $\binom{n}{i}$ possible index sets where exactly i coordinates of w differ from v, and $q-1$ possible values for each of these coordinates. Adding these numbers for $0 \leq i \leq r$ gives $V_q(n, r)$, the number of vectors $w \in GF(q)^n$ with $d(v, w) \leq r$. This is the same as the number of vectors in a ball of radius r around the center v. □

Definition 15.21. Let $d,\ n \in \mathbb{N}$ and $d \leq n$. Then we define $A_q(n,\ d)$ to be the largest integer M such that a code C over $GF(q)$ of size M, length n and minimum distance $\geq d$ exists. ♢

The following Theorem gives a lower bound for $A_q(n, d)$:

Theorem 15.22. (*Sphere-covering bound*)

$$A_q(n,\ d) \geq \frac{q^n}{V_q(n,\ d-1)}.$$

Proof. Let C be a code of length n, minimum distance of at least d and $|C| = A_q(n,\ d)$. We can assume that, for each vector $v \in GF(q)^n$, there is at least one codeword c such that $d(v, c) < d$. Otherwise, we could add v as a codeword to the code while preserving the length n and the minimum distance d. Hence the union of balls of radius $d-1$

having their center at some codeword covers the whole of $GF(q)^n$. The number of vectors in that union is at most $A_q(n,\ d) \cdot V_q(n,\ d-1)$, which implies the sphere-covering bound. □

Note that the above argument does not show equality, since vectors can be contained in several balls.

The following theorem gives a bound for the existence of a *linear code* of dimension k and minimum distance d.

Theorem 15.23. (*Gilbert-Varshamov bound*) *Let $n \geq 2$, $k \leq n$ and $d \geq 2$ be integers such that*

$$V_q(n-1,\ d-2) < q^{n-k}.$$

Then there exists a linear $[n,k]$ code over $GF(q)$ with minimum distance $\geq d$.

Proof. The assumption $q^{n-k} > V_q(n-1,\ d-2)$ ensures that we can find n vectors in $GF(q)^{n-k}$ such that any $d-1$ of them are linearly independent (see [**Rot06**] for more details). We write these vectors into the columns of a $(n-k) \times n$ parity-check matrix H. The dimension of the associated code is at least $n-(n-k) = k$. Furthermore, the distance between two different codewords cannot be less than d, since otherwise a codeword $c \in C$ exists with $wt(c) \leq d-1$, and so $d-1$ columns of H are linearly dependent, a contradiction. □

An upper bound for $A_q(n,d)$ is given by the *Hamming bound*:

Proposition 15.24. (*Hamming bound or sphere-packing bound*) *For $d = 2e+1$, we have*

$$A_q(n,d) \leq \frac{q^n}{V_q(n,e)}.$$

Proof. Suppose a code of length n, minimal distance $d = 2e+1$ and size $A_q(n,d)$ is given. Let $B_e(v)$ be the set of vectors w with $d(v,w) \leq e$, i.e., a ball of radius e around v. The cardinality of $B_e(c)$ is $V_q(n,e)$. The union of the balls $B_e(c)$ over all codewords c is disjoint and has $A_q(n,d) \cdot V_q(n,e)$ elements. Therefore, we obtain

$$A_q(n,d) \cdot V_q(n,e) \leq q^n.$$

This yields the assertion. □

A code is called *perfect* if it attains the Hamming bound.

Example 15.25. Let $q=2$, $n=7$ and $d=3$. Then the sphere-covering bound is

$$\frac{2^7}{V_2(7,2)} = \frac{2^7}{\binom{7}{0}+\binom{7}{1}+\binom{7}{2}} = \frac{128}{29} \approx 4.4,$$

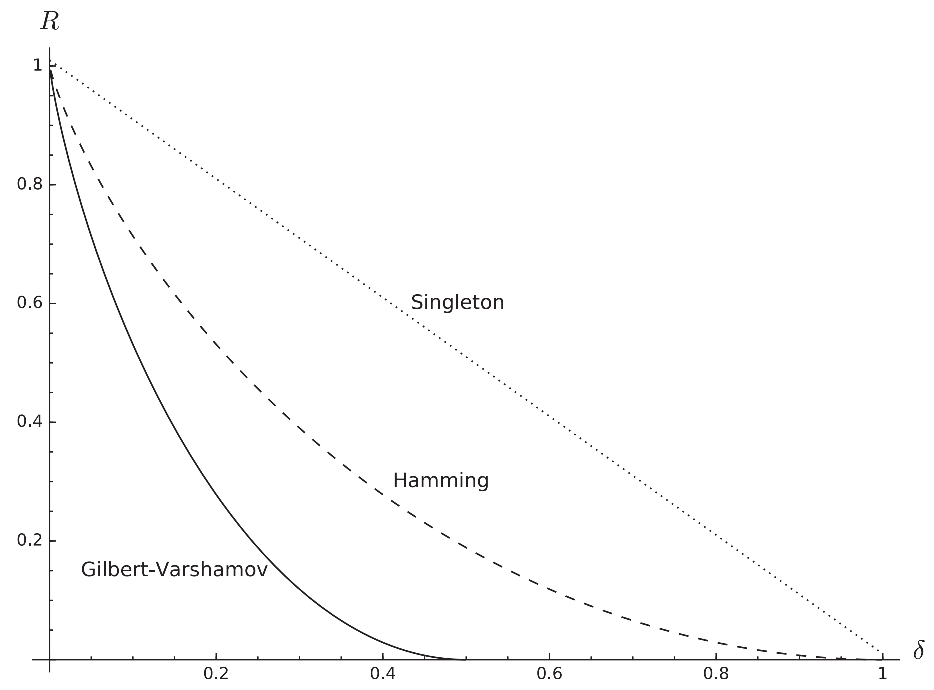


Figure 15.2. Asymptotic bounds on the information rate $R = \frac{k}{n}$ against the relative minimum distance $\delta = \frac{d}{n}$ for codes over $GF(2)$. Singleton and Hamming are upper bounds and Gilbert-Varshamov is a lower bound.

and hence there is a code with at least 5 codewords and the above parameters. Next, we consider the Gilbert-Varshamov bound:

$$V_2(6,1) = \binom{6}{0} + \binom{6}{1} = 7 < 2^3 = 2^{7-4}.$$

This implies that a linear $[7,4]$ code with minimum distance ≥ 3 exists. Since the $[7,4,3]$ Hamming code has 16 codewords, the code attains the Gilbert-Varshamov bound. The Hamming bound is

$$\frac{2^7}{V_2(7,1)} = \frac{128}{\binom{7}{0} + \binom{7}{1}} = 16.$$

Therefore, the maximum size of a code of length 7 over $GF(2)$ with a minimum distance of at least 3 is 16. We conclude that the Hamming $[7,4,3]$ code is perfect. ◊

We return the problem at the beginning of this section on good codes of length n. The relative minimum distance $\delta = \frac{d}{n}$ and the information rate $R = \frac{k}{n}$ cannot be close to 1 at the same time. The bounds for $n \to \infty$ are shown in Figure 15.2.

15.3. Goppa Codes

Since our main objective is to explain the McEliece cryptosystem, we continue with Goppa codes. Textbooks on coding theory would deal with (generalized) Reed-Solomon (RS) and Bose-Chaudhuri-Hocquenghem (BCH) codes first, but this is beyond the scope of this book.

General Goppa codes are geometric codes which are constructed using curves over finite fields. For our purposes, we only consider classical irreducible binary Goppa codes. They are defined by a polynomial in one variable over $GF(2^m)$.

We fix an integer $m \geq 3$ for a code C over $GF(2^m)$. Furthermore, we fix an integer t such that $2 \leq t \leq \frac{2^m-1}{m}$. The code is designed to have a minimum distance of $d(C) = 2t+1$ and to fix up to t errors. It is common to set the length to $n = 2^m$, but $mt + 1 \leq n \leq 2^m$ gives greater flexibility (see [**BLP08**]). A classical Goppa code Γ over $GF(2)$ is a *subfield code* of C, i.e., $\Gamma = C \cap GF(2)^n$.

Furthermore, let $a_1, \ldots, a_n$ be distinct *code locator* elements of $GF(2^m)$. If $n = 2^m$, then all elements of the field are code locators. Finally, we fix a random irreducible polynomial $g \in GF(2^m)[x]$ of degree t.

Example 15.26. Typical choices for cryptographic applications are $10 \leq m \leq 13$ and $50 \leq t \leq 150$, for example $m = 10$, $n = 2^m = 1024$ and $t = 50$. ◊

Below, we consider residue classes in $GF(2^m)[x]/(g(x))$. Since we assumed that g is irreducible, the latter ring is a *field*. Hence polynomials $f \in GF(2^m)[x]$ are invertible modulo g if f is not zero or a multiple of g. In particular, the residue class

$$\frac{1}{x - a_i} \bmod g$$

exists and can be represented by a polynomial in $GF(2^m)[x]$ of degree less than t.

Definition 15.27. Using the above parameters, define a linear $[n, t]$ code C over $GF(2^m)$ by

$$C = \left\{ (c_1, \ldots, c_n) \in GF(2^m)^n \mid \sum_{i=1}^{n} c_i \, \frac{1}{x - a_i} \equiv 0 \bmod g \right\}.$$

The definition of C depends on the order of the code locator elements, but re-ordering gives an equivalent code. ◊

Multiplying the above equation by $h(x) = \prod_{i=1}^{n}(x - a_i)$ eliminates the denominators $x - a_i$. Note that $h(x)$ is invertible modulo g. For $n = 2^m$, one has $h(x) = x^n - x$, since the roots of $x^n - x$ are precisely the elements of the field $GF(2^m)$ (see Proposition 4.73).

The code C is the kernel of the syndrome map

$$Syn : \; GF(2^m)^n \to GF(2^m)[x]/(g(x))$$

that sends $(c_1, \ldots, c_n)$ to $\sum_i c_i \frac{h}{x-a_i} \mod g$. We may represent the syndrome by a polynomial of degree less than t. This yields t parity-check equations in n variables and C is therefore a $[n, n-t]$ code over $GF(2^m)$, assuming that the equations are linearly independent (see Remark 15.28 below).

Remark 15.28. One can show ([**Rot06**] Section 5.1 and Problem 5.11) that C is a *Generalized Reed-Solomon* (GRS) code with the following parity-check matrix over $GF(2^m)$:

$$H = \begin{pmatrix} 1 & \ldots & 1 \\ a_1 & \ldots & a_n \\ & \ldots & \\ a_1^{t-1} & \ldots & a_n^{t-1} \end{pmatrix} \begin{pmatrix} \frac{1}{g(a_1)} & \ldots & 0 \\ & \ldots & \\ & \ldots & \\ & \ldots & \\ 0 & \ldots & \frac{1}{g(a_n)} \end{pmatrix} = \begin{pmatrix} \frac{1}{g(a_1)} & \ldots & \frac{1}{g(a_n)} \\ \frac{a_1}{g(a_1)} & \ldots & \frac{a_n}{g(a_n)} \\ & \ldots & \\ \frac{a_1^{t-1}}{g(a_1)} & \ldots & \frac{a_n^{t-1}}{g(a_n)} \end{pmatrix}.$$

The first t columns of the first matrix have a *Vandermonde* form and are thus nonsingular. The second matrix is a nonsingular diagonal matrix. This shows that the rows of H are linearly independent, so that C is an $[n, n-t]$ code. Using Proposition 15.18, one can also show that C is MDS, i.e., an $[n, n-t, t+1]$ code. $\diamondsuit$

In principle, GRS codes can be used for encryption. However, several proposals to use GRS codes turned out to be insecure while Goppa codes are still unbroken.

Definition 15.29. Let C be a code over $GF(2^m)$ as in Definition 15.27. We define the corresponding classical irreducible binary *Goppa code* Γ over $GF(2)$ to be the *subfield code* of C, i.e.,

$$\Gamma = \left\{ (c_1, \ldots, c_n) \in GF(2)^n \mid \sum_{i=1}^{n} c_i \frac{1}{x - a_i} \equiv 0 \mod g \right\}. \qquad \diamondsuit$$

Since $h(x) = \prod_{i=1}^{n} (x - a_i)$ is invertible modulo g, one has $c \in \Gamma$ if and only if

$$\sum_{i=1}^{n} c_i \frac{h}{x - a_i} \mod g \equiv 0.$$

The parity-check matrix of Γ is essentially the matrix H in Remark 15.28, but now each element is viewed as a column of m elements of $GF(2)$.

Proposition 15.30. *Let Γ be a classical irreducible binary Goppa code with the above parameters. Then Γ is a $[n, \geq n - mt, \geq 2t+1]$ code.*

Proof. C is defined by t parity-check equations over $GF(2^m)$, and this corresponds to mt equations over $GF(2)$. Since the equations may not be linearly independent, the dimension of Γ is $\geq n - mt$.

We follow [**Ber11**] and prove that the distance of codewords is at least $2t+1$. It is sufficient to show that the weight of all codewords is $\geq 2t+1$. Let $c \in \Gamma$ be a nonzero codeword. We define the following polynomials over $GF(2^m)$:

$$f = \prod_{i:\ c_i=1} (x - a_i) \text{ and } f_i = \frac{f}{x - a_i} \text{ for } c_i = 1.$$

Note that the degree of f is equal to the weight of c. Let $D(f)$ be the formal derivative of f (see Definition 4.55). One has

$$D(f) = D((x - a_i)f_i) = f_i + (x - a_i)D(f_i)$$

for $i \in \{1, \dots, n\}$ with $c_i = 1$. A recursive application gives

$$D(f) = \sum_{i:\ c_i=1} f_i.$$

Multiplying this equation with the polynomial $\frac{h}{f}$ yields

$$\frac{h\,D(f)}{f} = \sum_{i:\ c_i=1} \frac{h}{x - a_i} = \sum_{i=1}^{n} c_i \,\frac{h}{x - a_i}\,.$$

By assumption, we have $c \in \Gamma$, and hence $\sum_{i=1}^{n} c_i \frac{h}{x-a_i} \equiv 0 \mod g$. Therefore, $\frac{h\,D(f)}{f}$ is a multiple of g. Since $g(a_i) \neq 0$ for all $i = 1, \dots, n$, the polynomials g and $\frac{h}{f}$ are relatively prime, which implies $g \mid D(f)$. Now, polynomials over $GF(2^m)$ have a surprising property (see Exercise 7):

- All elements in $GF(2^m)$ are squares and
- All polynomials over $GF(2^m)$ can be written as $\alpha^2 + x\beta^2$ with $\alpha,\ \beta \in GF(2^m)[x]$.

Write $f = \alpha^2 + x\beta^2$ with $\alpha, \beta \in GF(2^m)[x]$. By construction, f has only simple roots and is not a square. Hence $\beta \neq 0$ and

$$D(f) = 2\alpha D(\alpha) + \beta^2 + 2x\beta D(\beta) = \beta^2.$$

Since g is irreducible and $g \mid D(f)$ (see above), we obtain $g \mid \beta$. We conclude that the degree of β is at least t and the degree of $f = \alpha^2 + x\beta^2$ is at least $2t + 1$. This proves $wt(c) = \deg(f) \geq 2t + 1$. □

Example 15.31. Let $m = 4$, $t = 2$ and $n = 16$; then $n - mt = 8$ and $d = 2t + 1 = 5$. We want to construct a $[16, 8, 5]$ Goppa code over $GF(2)$ and use SageMath for the computations. The field $GF(16)$ is given by

$$GF(2)[z]/(z^4 + z + 1)$$

and its elements $a_1, \dots, a_{16} \in GF(16)$ are represented by binary polynomials in the variable z of degree < 4. We choose an irreducible polynomial $g \in GF(16)[x]$ of degree 2:

$$g(x) = x^2 + z^2x + z.$$

```
sage: K.<z>=GF(2^4, name='z', modulus=x^4+x+1)
sage: PR.<x>=K[]
sage: g=x^2+z^2*x+z; g.is_irreducible()
True
```

The factor ring $GF(16)[x]/(g(x))$ is a field with $16^2 = 256$ elements.

```
sage: Rmodg=PR.quotient_ring(g)
```

The elements

$$\frac{1}{x - a_i} \bmod g, \; a_i \in GF(16)$$

can be represented by polynomials in $GF(16)[x]$ of degree ≤ 1.

```
sage: arr=[]
sage: for a in K.list():
          arr.append(1/Rmodg(x-a))
```

The array `arr` contains all elements $\frac{1}{x-a} \bmod g$ with $a \in GF(16)$. Their coefficients with respect to the standard basis $\{1, x\}$ define a 2×16 parity check matrix H_{16} of the code C over $GF(16)$:

```
sage: H16=matrix(K,2,16)
sage: for i in range(0,2):
          for j in range(0,16):
              H16[i,j]=list(arr[j])[i]
```

In our example, one obtains

$$H_{16} = \begin{pmatrix} z & z^3+z & 0 & z^3 & \dots & \dots & z^3+z^2+1 \\ z^3+1 & z+1 & z^3+1 & z^3+z^2+z+1 & \dots & \dots & z^2+z \end{pmatrix}.$$

H_{16} defines a $[16, 14]$ code over $GF(16)$. The parity-check matrix of the subfield code Γ *over* $GF(2)$, denoted by H, is obtained by replacing the coefficients in $GF(16)$ by a column vector of length 4 over $GF(2)$, using the standard basis $\{z^3, z^2, z, 1\}$ of $GF(16)$ over $GF(2)$.

For example, the lower left entry z^3+1 is replaced by the column vector $(1, 0, 0, 1)^T$.

```
sage: H=matrix(GF(2),8,16)
sage: for i in range(0,2):
          for j in range(0,16):
              H16[i,j]=list(arr[j])[i]
              hbin=bin(eval(H16[i,j]._int_repr()))[2:]
              hbin='0'*(4-len(hbin))+hbin; hbin=list(hbin);
              H[4*i:4*(i+1),j] = vector(map(GF(2),hbin));
```

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

H is the parity-check matrix of a Goppa code Γ. By solving the linear system of equations $vH^T = 0$ and performing elementary row operations, we get the generator matrix G of Γ in systematic form.

```
sage: G=H.right_kernel().basis_matrix()
```

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Proposition 15.30 shows that Γ is a $[16, \geq 8, \geq 5]$ code. Since the rank of G is 8 and codewords of weight 5 exist, Γ is a $[16, 8, 5]$ code and can decode 16-bit words having at most two errors. There are $2^8 = 256$ different syndromes and their coset leader gives the error vector. $\Diamond$

Syndrome decoding of general $[n, k]$ codes is inefficient if n is large. Although there are more effective approaches (*information-set decoding*, see below), decoding is a hard problem for general codes. Now, classical binary irreducible Goppa codes have the useful property that they possess an efficient decoding algorithm, even for large dimensions, because of their special structure. Below, we describe the *Patterson algorithm* [**Pat75**]. In the next Section 15.4, we explain how Goppa codes are leveraged to construct a public-key cryptosystem.

Algorithm 15.1 Decoding Goppa Codes using Patterson's Algorithm

```
Input: Code locator elements $a_1, \dots, a_n$, Goppa polynomial g of degree $t$, erroneous
    word $w = (w_1, \dots, w_n)$ with up to $t$ errors.
Output: Codeword $c$
 1: Compute $Syn(w) \mod g$ and $T = (Syn(w) \mod g)^{-1}$.
 2: if $T \equiv x \mod g$ then
 3:     $\sigma = x$
 4: else
 5:     Compute $R = \sqrt{T + x}$.
 6:     Run Extended Euclidean Algorithm on input g and $R$.
 7:     Get $(\alpha, \beta)$ such that $\alpha = \beta R + yg$ and $\deg(\alpha) \leq \frac{t}{2}$, $\deg(\beta) \leq \frac{t-1}{2}$.
 8:     $\sigma = \alpha^2 + x\beta^2$
 9: end if
10: Compute roots of $\sigma$. Error vector $e$ has a 1 in coordinate $i$ if $a_i$ is a root of $\sigma$.
11: $c = w + e$
12: return $c$
```

Suppose that all of the parameters of a Goppa code are known and a word $w = (w_1, \dots, w_n) \in GF(2)^n$ with at most t errors is received. We want to decode w and proceed similarly as in the proof of Proposition 15.30.

Let $f = \prod_{i:w_i=1}(x - a_i)$ and $f_i = \frac{f}{x-i}$. Then $D(f) = \sum_{i:\ w_i=1} f_i$ and

$$Syn(w) = \sum_{i:\ w_i=1} \frac{1}{x - a_i} = \frac{D(f)}{f} \mod g.$$

If $Syn(w) \equiv 0 \mod g$, then w is already a codeword. Now assume that $Syn(w) \not\equiv 0 \mod g$. We want to find the *error locator polynomial* $\sigma = \prod_{i:e_i=1}(x - a_i)$ of degree $\leq t$ with the same syndrome as f:

$$Syn(w) = \frac{D(f)}{f} = \frac{D(\sigma)}{\sigma} \mod g.$$

The roots of σ correspond to the nonzero entries of the error vector e. Set $\sigma = \alpha^2 + x\beta^2$ with $\alpha, \beta \in GF(2^m)[x]$. The polynomials α and β are uniquely determined and $\deg(\alpha) \leq \frac{t}{2}$ and $\deg(\beta) \leq \frac{t-1}{2}$. We now compute the polynomials α and β from the given syndrome $Syn(w)$. One has $D(\sigma) = \beta^2$ and hence

$$Syn(w)(\alpha^2 + x\beta^2) = \beta^2 \mod g.$$

Letting $T = Syn(w)^{-1} \mod g$ we obtain

$$(\alpha^2 + x\beta^2) = T\beta^2 \mod g \iff \alpha^2 = \beta^2(T - x) \mod g.$$

If $T \equiv x \mod g$, then set $\alpha = 0, \beta = 1$ and

$$\sigma = \alpha^2 + x\beta^2 = x.$$

In this case 0 is the only root and the i-th bit is erroneous, where $a_i = 0 \in GF(2^m)$.

Otherwise, compute $R = \sqrt{T - x} \mod g$. We note that a unique square root exists, since $GF(2^m)[x]/(g)$ is a finite field of characteristic 2 (see Exercise 6). Huber's observation ([**Hub96**], see Exercise 8) provides an efficient algorithm to compute the square root in a binary field. This gives

$$\alpha = \beta R \mod g.$$

We lift the residue class R to a polynomial over $GF(2^m)$ of degree $< t$. The requested solution (α, β) can be found by performing several iterations of the Extended Euclidean Algorithm in $GF(2^m)[x]$ on inputs g and R until $\deg(\alpha) \leq \frac{t}{2}$ and $\deg(\beta) \leq \frac{t-1}{2}$. The algorithm outputs polynomials α, β and y in $GF(2^m)[x]$ such that

$$\alpha = \beta R + yg.$$

It is necessary to stop the Euclidean Algorithm midway as soon as the degree of a remainder is $\leq \frac{t}{2}$. Further iterations decrease the degree of α (until $\deg(\alpha) = 0$), but increase the degree of β above the $\frac{t-1}{2}$ limit. Then the error polynomial is

$$\sigma = \alpha^2 + x\beta^2.$$

Finally, the roots of σ correspond to the error positions of w. A brute-force search in $GF(2^m)$ can be performed, but there are also other methods. The Patterson algorithm is quite efficient and the root extraction turns out to be the most expansive part in the decoding.

Example 15.32. Consider the Goppa code from Example 15.31. We encode $(1, 1, 0, 1, 0, 0, 1, 0)$ and obtain the codeword

$$c = (1, 1, 0, 1, 0, 0, 1, 0)\, G = (1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0).$$

Adding two errors at the third and seventh positions yields the word

$$w = (1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0).$$

We apply Patterson's algorithm to decode w. First, we compute the syndrome (which depends on the ordering of $a_1, \ldots, a_{16} \in GF(16)$):

$$Syn(w) = \sum_{i:\ w_i=1} \frac{1}{x - a_i} \mod g.$$

We re-use our array `arr` of elements $\frac{1}{x-a_i} \mod g$ (see Example 15.31).

```
sage: c=vector(GF(2),[1,1,0,1,0,0,1,0])*G
sage: e=vector(GF(2),[0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0])
sage: w=c+e
sage: syn=w*vector(arr); syn
(z^2 + z)*xbar + 1
```

Note that SageMath writes `xbar` for the residue class $x \mod g$. We obtain

$$Syn(w) = (z^2 + z)x + 1 \mod g$$

and let

$$T = \frac{1}{Syn(w)} \equiv z^3 x + (z^3 + z + 1) \mod g.$$

Next, we have to compute the square root of $T + x = (z^3 + 1)x + (z^3 + z + 1)$. Since $GF(16)[x]/(g(x))$ is a binary field with 256 elements, one has (see Exercise 6)

$$R = \sqrt{T + x} = (T+x)^{128} = ((z^3+1)x + (z^3+z+1))^{128} \equiv (z^3+z^2)x + (z^2+z+1) \mod g.$$

Note that the root R can be computed more efficiently (see Exercise 8).

```
sage: T=1/(w*vector(arr))
sage: T - Rmodg(x)
(z^3 + 1)*xbar + z^3 + z + 1
sage: R=(T - Rmodg(x))^128; R
(z^3 + z^2)*xbar + z^2 + z + 1
```

Finally, we have to find α, β with $\alpha = \beta R \mod g$. In our example, the degrees have to satisfy $\deg(\alpha) \leq 1$ and $\deg(\beta) = 0$. So we simply define α as the lift of R to $GF(16)[x]$ and set $\beta = 1$. The error polynomial is

$$\sigma = \alpha^2 + x\beta^2 = ((z^3 + z^2)x + (z^2 + z + 1))^2 + x = (z^3 + z^2 + z + 1)x^2 + x + (z^2 + z).$$

The error locator polynomial $\sigma \in GF(16)[x]$ is of degree $t = 2$ and its roots are z^2 and $z^3 + z^2$.

```
sage: a=(z^3+z^2)*x+z^2+z+1; b=1
sage: sigma=a*a+x*b*b;sigma
(z^3 + z^2 + z + 1)*x^2 + x + z^2 + z
sage: sigma.factor()
(z^3 + z^2 + z + 1) * (x + z^2) * (x + z^3 + z^2)
```

We fixed an ordering of the elements in $GF(16)$, and in our example, the roots are the third and the seventh field elements. Hence the word w has errors at positions 3 and 7 and we recover the codeword by adding the error vector $e = (0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0)$.

```
sage: i=0; e=vector(GF(2),16)
sage: for k in list(K):
          if ((sigma.subs(x=k))==0):
              e[i]=1
          i=i+1
sage: print(e)
(0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Finally, we verify that e has the same syndrome as w.

```
sage: e*vector(arr)
(z^2 + z)*xbar + 1
```

Since n is small, classical syndrome decoding, i.e., without using the Goppa code structure, would also work in this example. One finds that the coset leader of $w + \Gamma$ is the error vector $e = (0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0)$, since $wt(e) = 2$ and the syndrome of both w and e is

$$wH^T = eH^T = (0,0,0,1,0,1,1,0).$$

```
sage: w*H.transpose()
(0, 0, 0, 1, 0, 1, 1, 0)
sage: e*H.transpose()
(0, 0, 0, 1, 0, 1, 1, 0)
```

15.4. McEliece Cryptosystem

The idea of code-based encryption is quite simple: choose a code, encode the plaintext, change the codeword at some positions and take the result as ciphertext. The challenge is to make this a *trapdoor one-way function* and to provide an efficient decoding algorithm that uses a secret structure of the code. The McEliece cryptosystem is based on Goppa codes, but the public generator matrix is modified in order to conceal the Goppa code structure. Decoding should in general be hard, but is easy when the underlying secret Goppa code is known.

First, we explain the classical McEliece cryptosystem.

Definition 15.33. Suppose the Goppa code parameters n, m and t are given such that $m \geq 3$, $n \leq 2^m$ and $2 \leq t < \frac{n}{m}$. Set $k = n - mt$ and $d = 2t + 1$. Then the McEliece cryptosystem is defined by:

- The plaintext space $\mathcal{M} = \{0,1\}^k$ and the ciphertext space $\mathcal{C} = \{0,1\}^n$.
- A uniform random secret key consisting of an invertible $k \times k$ matrix S over $GF(2)$, an $n \times n$ permutation matrix P over $GF(2)$, distinct elements $a_1, \dots, a_n \in GF(2^m)$, an irreducible polynomial $g \in GF(2^m)[x]$ of degree t and the $k \times n$ generator matrix G of the associated Goppa code Γ. A new key is chosen if the dimension of Γ is not $n - mt$.
- A public key, defined by the $k \times n$ matrix $G_1 = SGP$ over $GF(2)$.
- An encryption algorithm that takes a plaintext $x \in \{0,1\}^k$ as input, generates a random binary string e of length n and weight t and outputs the ciphertext
$$y = \mathcal{E}_{pk}(x) = xG_1 + e.$$
- A decryption algorithm that takes a ciphertext $y \in \{0,1\}^n$ as input, computes $y_1 = yP^{-1}$ and efficiently decodes y_1 to the codeword c by using the secret Goppa code

structure. The plaintext x is obtained by solving the linear system of equations $xSG = c$. $\Diamond$

We show that decryption is correct: let $y = xG_1 + e = xSGP + e$ be a ciphertext; then

$$y_1 = yP^{-1} = xSGPP^{-1} + eP^{-1} = xSG + eP^{-1}.$$

Since P is a permutation, eP^{-1} has weight t. Hence y_1 can be decoded to the codeword $c = (xS)\,G$ of the secret Goppa code having the generator matrix G. The original information word that corresponds to the codeword c is xS, and multiplication by S^{-1} recovers the plaintext x.

Example 15.34. We use the Goppa code from Example 15.31 and choose random secret matrices S and P. S is an invertible 8×8 matrix over $GF(2)$:

$$S = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

P describes a permutation of $GF(2)^{16}$ and its columns are unit vectors e_i:

$$P = \begin{pmatrix} \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert \\ e_{11} & e_{13} & e_3 & e_8 & e_{15} & e_5 & e_2 & e_1 & e_9 & e_7 & e_6 & e_{16} & e_4 & e_{10} & e_{12} & e_{14} \\ \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert & \vert \end{pmatrix}.$$

We use the generator matrix G of the Goppa code from Example 15.31. The matrix $G_1 = SGP$ generates a $[16, 8, 5]$ code and is public:

$$G_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

Suppose we want to encrypt the plaintext $x = (0, 1, 1, 1, 0, 0, 1, 1)$. We choose the following random error vector of weight 2:

$$e = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0).$$

The ciphertext is

$$y = xG_1 + e = (0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1).$$

For decryption, we use the matrices P, S and the Goppa code. First, compute

$$y_1 = yP^{-1} = (0,1,0,0,0,1,0,1,1,1,0,0,0,1,1,1).$$

Now we use the Goppa code structure to find the error vector eP^{-1} and decode y_1 to the codeword

$$c = (0,1,0,0,0,1,0,1,1,0,0,0,1,1,1,1)$$

(Exercise 10). Finally, one solves the linear system of equations $xSG = c$ and finds the plaintext $x = (0,1,1,1,0,0,0,1,1)$. $\diamondsuit$

The security of the McEliece cryptosystem depends on how hard it is to decode a vector with t errors for a code given by the generator matrix $G_1 = SGP$. Since S and P are secret, the matrix G and the Goppa code structure cannot be used by an adversary.

The best known attacks are based on *information-set decoding*. The basic idea is quite simple: choose k columns of the public $k \times n$ generator matrix G_1 such that the resulting $k \times k$ matrix is invertible. The indices of the chosen columns define an *index set* I and we denote the resulting $k \times k$ submatrix by G_I. Suppose we want to decode a vector y of length n. The information word x and the codeword $c = xG_1$ are unknown. Let y_I be the subvector of length k defined by the index set I. We compute

$$x_I = y_I G_I^{-1}.$$

If the vector y coincides with the codeword c at the index positions I, i.e., if all errors are outside I, then $x_I = x$ and we have successfully decoded y. This follows from basic linear algebra: the linear system of equations $xG_1 = c$ is overdetermined with k variables and $n > k$ equations. Choosing k linearly independent equations, i.e., an invertible $k \times k$ submatrix G_I of G_1, suffices to find x.

However, it is unlikely that y is error-free on a random index set I: the probability that a randomly selected index set is not affected by any errors is

$$\frac{n-t}{n} \cdot \frac{n-t-1}{n-1} \cdots \frac{n-t-k+1}{n-k+1} < \left(\frac{n-t}{n}\right)^k = \left(1 - \frac{t}{n}\right)^k.$$

Hence the number of attempts necessary to find a suitable information set I is exponential in k. There are many improvements to this basic scheme, but the number of guesses remains exponential in the number of errors added.

Example 15.35. We return to Example 15.34 and want to decode the ciphertext y without the Goppa code structure, using information-set decoding. We choose the index set

$I = \{4, 5, 6, 7, 8, 9, 10, 11\}$, extract columns 4 – 11 from G_1 and invert the matrix:

$$G_I = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \; G_I^{-1} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Now we hope to compute the plaintext:

$$x_I = y_I G_I^{-1} = (1, 1, 0, 1, 0, 1, 0, 1)\, G_I^{-1} = (0, 1, 1, 1, 0, 0, 1, 1).$$

Indeed, we have $x = x_I$ and have successfully computed the plaintext. The attack works because the error positions (2 and 14) lie outside the information set I.

However, if an adversary chooses the index set $I = \{1, 2, 3, 4, 5, 6, 7, 8\}$, the result is $x_I = (0, 0, 0, 1, 1, 1, 0, 1)$. They can verify whether x_I is correct by computing $x_I G$ and subtracting (i.e., adding modulo 2) the ciphertext y:

$$x_I\, G_1 + y = (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1).$$

Since the weight is 6, the result is not a valid error vector and the attack has failed. $\diamondsuit$

There are several improvements of McEliece's original system. Firstly, the public generator matrix $G_1 = SGP$ can be transformed into the *systematic* form $G_1' = (I_k \mid G_2)$ by elementary row operations and (possibly) column permutations. These operations correspond to matrix multiplications from the left (row operations) and from the right (column permutations). One obtains $G_1' = S'GP'$, so the underlying Goppa code with the generator matrix G remains unchanged.

The advantage of the systematic form $(I_k \mid G_2)$ is that the public key can be compressed to the $k \times (n - k)$ submatrix G_2, since the identity matrix is a constant part of any matrix of this form.

Now, suppose the public generator matrix is in systematic form $(I_k \mid G_2)$. Then encryption of a plaintext x yields the ciphertext

$$y = x(I_k \mid G_2) + e = (x \,||\, G_2\, x) + e.$$

We observe that the plaintext x – only slightly disturbed by at most t error bits – is a part of the ciphertext! Although this is not a problem (or even desirable) for channel coding, it looks alarming with respect to the security of the McEliece cryptosystem. In fact, the original system is not EAV-, CPA- or CCA2-secure.

However, it is assumed that the McEliece encryption function is *one-way* and it is hard to recover *all bits* of a uniform random plaintext from a given ciphertext. Furthermore, there are techniques to reduce the use of partial information, and a modification of the original McEliece cryptosystem can achieve CCA2 security in the random oracle model. The idea of *Pointcheval's generic conversion* [**Poi00**] is to encrypt a uniform random string r of length k instead of the plaintext x. The error vector is obtained by applying a cryptographic hash function H to $(x\|r')$, where r' is a uniform random string of length k':

$$e = H(x\|r'), \ \ y_1 = rG_1 + e.$$

e needs to be transformed into an error vector of length n and weight t, but we can skip the details here. Let R be a pseudorandom generator that takes a key (or seed) of length k as input and outputs a string of length $k + k'$. Then set

$$y_2 = (x\|r') \oplus R(r).$$

The ciphertext is defined by $y = (y_1\|y_2)$.

Decryption takes $(y_1\|y_2)$ as input and recovers r from y_1 as in the original McEliece cryptosystem. Then, one obtains x and r':

$$(x\|r') = y_2 \oplus R(r).$$

Before outputting the plaintext x, the integrity is checked using r'. For this purpose, the resulting error vector $rG_1 + y_1$ is compared to the error vector derived from $H(x\|r')$. If they do not match, then an error code is returned. Otherwise, the plaintext x is output.

An adversary can still obtain large parts of r from the ciphertext y_1. However, they cannot exploit this information, unless they have the complete key r, which is very unlikely. Furthermore, access to a decryption oracle does not help decrypt a given challenge ciphertext.

Finally, we want to explain the *Niederreiter cryptosystem*, which uses *syndromes* instead of erroneous codewords. An advantage is that the Niederreiter cryptosystem has shorter ciphertexts than the McEliece cryptosystem.

Definition 15.36. Suppose the Goppa code parameters n, m and t are given such that $m \geq 3$, $n \leq 2^m$ and $2 \leq t < \frac{n}{m}$. Set $k = n - mt$ and $d = 2t + 1$. The Niederreiter cryptosystem is defined as follows:

- The plaintext space contains the binary strings of length n and weight t. The ciphertext space is $\mathcal{C} = \{0, 1\}^{n-k}$.
- The secret key is chosen uniformly at random and consists of an invertible $(n - k) \times (n - k)$ matrix S over $GF(2)$, an $n \times n$ permutation matrix P over $GF(2)$, distinct elements $a_1, \dots, a_n$ of the field $GF(2^m)$, an irreducible polynomial $g \in GF(2^m)[x]$ of degree t and the $(n - k) \times n$ parity-check matrix H of the associated Goppa code Γ. A new key is chosen if the dimension of Γ is not $n - mt$.
- The public key is the $(n - k) \times n$ matrix $H_1 = SHP$.

- The encryption algorithm takes a plaintext $x \in \{0,1\}^n$ of weight t as input and outputs the ciphertext
$$y = \mathcal{E}_{pk}(x) = H_1 x^T.$$
- The decryption algorithm takes a ciphertext $y \in \{0,1\}^{n-k}$ as input and computes the column vector $syn = S^{-1}y$. Find a vector z such that $Hz^T = syn$ and decode z using Patterson's algorithm. This gives the error vector e of weight t with $He^T = syn$. The plaintext x is recovered by
$$x^T = P^{-1}e^T. \qquad \diamondsuit$$

The public key $H_1 = SHP$ is transformed into the systematic form $(I_{n-k}|H_2)$. This reduces the size of the public key to $k(n-k)$ bits.

We explain the correctness of the Niederreiter cryptosystem: $H(Px^T)$ is the syndrome of Px^T. Thus the ciphertext $y = SHPx^T$ is a transformed syndrome. Computing $S^{-1}y$ gives the syndrome $H(Px^T)$. Syndrome decoding recovers the error vector $e^T = Px^T$, and the plaintext is $x^T = P^{-1}e^T$.

Example 15.37. Consider the Goppa code and the matrices H, S and P in Examples 15.31 and 15.34. Suppose we want to encrypt the plaintext
$$x = (0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0)$$
of weight $t = 2$ with the Niederreiter cryptosystem. The ciphertext is
$$y = SHPx^T = (0,0,1,0,1,1,1,0)^T.$$
For decryption, we compute the syndrome
$$syn = S^{-1}y = (0,0,1,1,1,1,0,1)^T.$$
The linear system of equations $Hz^T = syn$ is underdetermined. The affine solution space is of dimension 8 and one of the solutions is the vector
$$z = (1,0,1,1,0,1,0,1,0,0,0,0,0,0,0,0).$$
Syndrome decoding (see Exercise 12) yields the error vector
$$e^T = (1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)^T.$$
Finally, we recover the plaintext
$$x^T = P^{-1}e^T = (0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0)^T. \qquad \diamondsuit$$

Like the McEliece system, the plain Niederreiter cryptosystem is not CCA2 secure. This can be resolved by applying a CCA2-secure conversion, for example Pointcheval's generic conversion [**Poi00**] (see above), the Fujisaki-Okamoto transform [**FO13**] or the Kobara-Imai conversion [**NC12**].

Remark 15.38. The McEliece and the Niederreiter cryptosystems look differently, but they are based on the same decoding problem. Let G be the generator matrix and H the parity-check matrix of a Goppa code. Let $G_1 = SGP$ be the public key of the McEliece system and let S' be any invertible $(n-k) \times (n-k)$ matrix. Then $H_1 = S'H(P^{-1})^T$ satisfies $G_1 H_1^T = 0$, and H_1 is the parity-check matrix and the public key of the Niederreiter system of the same code. The McEliece ciphertext is an erroneous codeword and the Niederreiter ciphertext is a syndrome. Finding the nearest codeword is essentially the same as finding the coset leader of a syndrome. Therefore, the McEliece and the Niederreiter cryptosystems offer the same level of security. An adversary who can break one of them is also able to break the other. ♢

The McEliece and Niederreiter cryptosystems have been thoroughly investigated and are among the most promising candidates for post-quantum cryptography. McElices's key sizes ($n = 1024$, $k = 524$, $t = 50$) were broken 30 years after the original proposal with 2^{60} CPU cycles [**BLP08**]. For high security, D.J. Bernstein and T. Lange propose $[6960, 5413]$ Goppa codes with degree $t = 119$. A practical disadvantage is the large public-key key size of $k(n-k) = 8{,}373{,}911$ bits.

15.5. Summary

- Codes are used to detect and to correct errors when data is sent over noisy channels or stored on potentially unreliable media.
- Information words are encoded to codewords. Decoding of an erroneous codeword means finding the error vector, restoring the codeword and recovering the original data.
- There are bounds on the maximum number of codewords when the length and minimum distance of codewords is given.
- Syndrome decoding works in many practical applications, but decoding is a hard problem for random codes of large dimension.
- Goppa codes have an efficient decoding algorithm that also works for large dimensions.
- The McEliece cryptosystem is based on a code with a secret Goppa code structure. The ciphertext are erroneous codewords and the plaintext is recovered by decoding. The Niederreiter cryptosystem is similar to the McEliece scheme, but uses a parity-check matrix and syndromes for encryption.
- Code-based encryption with appropriate parameters is thought to be secure against attacks by quantum computers.

Exercises

1. The codewords of the parity code C of length n over $GF(q)$ are the words $(x_1, \dots, x_{n-1}, x_n)$ that satisfy $x_n = x_1 + \cdots + x_{n-1}$. Give the generator and the parity-check matrix of C. Show that C is a linear $[n, n-1, 2]$ MDS code.
2. Find the relationship between q-ary lattices and linear codes over $GF(q)$ for a prime q.
3. Let C be the linear $[8, 4]$ code with the following generator matrix over $GF(2)$:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

   (a) Show that G is also the parity-check matrix of C. Such a code is called *self-dual*.
   (b) Show that the minimum distance of C is $d = 4$.
   Hint: It is sufficient to show that every set of three columns of the parity-check matrix is linearly independent.
   (c) Decode the word received $y = (0, 1, 0, 0, 1, 1, 0, 0)$ using syndrome decoding.
4. Give the sphere-covering bound, the Gilbert-Varshamov bound and the Hamming bound for $n = 16$ and $d = 5$. Show that the Goppa code in Example 15.31 has maximal dimension.
5. Why is the formula $(a+b)^2 = a^2 + b^2$ (teacher's nightmare) true over binary fields $GF(2^m)$? What can be said about $(a + b)^n$ if n is a power of 2?
6. Why hold $a^{2^m} = a$ for every $a \in GF(2^m)$? Prove that

$$\sqrt{a} = a^{2^{m-1}}.$$

   Why is the square root unique? Show that

$$\sqrt{a+b} = \sqrt{a} + \sqrt{b} \text{ for all } a,\ b \in GF(2^m).$$

7. Show that any polynomial $f \in GF(2^m)[x]$ can be written as $f = \alpha^2 + x\beta^2$ with $\alpha,\ \beta \in GF(2^m)[x]$. How can α and β be computed?
8. Let $f(x) \in GF(2^m)[x]$ and suppose that $g(x) \in GF(2^m)[x]$ is irreducible. Why does a unique square root $\sqrt{f} \mod g(x)$ exist? Let $f = \alpha^2 + x\beta^2 \mod g(x)$ with $\alpha,\ \beta \in GF(2^m)[x]$. Show that

$$\sqrt{f} = \alpha + \sqrt{x}\,\beta \mod g(x).$$

Now suppose that $g = g_1^2 + xg_2^2$ with $g_1,\ g_2 \in GF(2^m)[x]$ and $1 = v_1g_1 + v_2g_2$ with $v_1,\ v_2 \in GF(2^m)[x]$. Show Huber's formulas [**Hub96**], [**Hub03**]:

$$\sqrt{x} = g_1\, g_2^{-1} \mod g(x),$$
$$\sqrt{x} = v_2g_1 + xv_1g_2 \mod g(x).$$

9. Refer to Example 15.32. Find the nearest codeword to the vector
$$(1,1,1,1,1,1,1,0,0,1,1,0,1,0,0,0).$$
10. Verify the results in Example 15.34 using SageMath.
11. What happens when the same plaintext is encrypted twice using the original McEliece cryptosystem? What information can an adversary gain from the ciphertexts? Now suppose that Pointcheval's generic conversion is used. What can now be said about the ciphertexts?
12. Decode the syndrome $(0,0,1,1,1,1,0,1)$ using the Goppa code in Example 15.31.

Bibliography

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, *PRIMES is in P*, Ann. of Math. (2) **160** (2004), no. 2, 781–793, DOI 10.4007/annals.2004.160.781. MR2123939

[Bar15] Gregory V Bard, *Sage for Undergraduates*, Vol. 87, American Mathematical Soc., 2015.

[Bar16] Elain Barker, *Recommendation for Key Management, Part 1: General*, Technical Report SP 800-57 Part 1 Revision 4, National Institute of Standards of Technology, 2016.

[BK15] Elaine Barker and John Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, Technical Report SP 800-90A Revision 1, National Institute of Standards and Technology, 2015.

[Bel06] Mihir Bellare, *New proofs for NMAC and HMAC: security without collision-resistance*, Advances in cryptology—CRYPTO 2006, Lecture Notes in Comput. Sci., vol. 4117, Springer, Berlin, 2006, pp. 602–619, DOI 10.1007/11818175_36. MR2422187

[BK03] Mihir Bellare and Tadayoshi Kohno, *A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications*, Advances in cryptology—EUROCRYPT 2003, Lecture Notes in Comput. Sci., vol. 2656, Springer, Berlin, 2003, pp. 491–506, DOI 10.1007/3-540-39200-9_31. MR2090438

[BR96] Mihir Bellare and Phillip Rogaway, *The exact security of digital signatures - How to sign with RSA and Rabin*, International conference on the theory and applications of cryptographic techniques, 1996, pp. 399–416.

[BR05] Mihir Bellare and Phillip Rogaway, *Introduction to Modern Cryptography*, UCSD CSE **207** (2005), 1–283.

[BR06] Mihir Bellare and Phillip Rogaway, *Code-based game-playing proofs and the security of triple encryption*, Advances in cryptology–eurocrypt. lncs, 2006, pp. 409–426.

[BB84] Charles H Bennett and Gilles Brassard, *Quantum Cryptography: Public Key Distribution and Coin Tossing*, Int. conf. on computers, systems and signal processing, Bangalore, India, 1984, pp. 175–179.

[BG07] Côme Berbain and Henri Gilbert, *On the Security of IV Dependent Stream Ciphers*, International workshop on fast software encryption, 2007, pp. 254–273.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters, *Attacking and defending the McEliece cryptosystem*, Post-quantum cryptography, Lecture Notes in Comput. Sci., vol. 5299, Springer, Berlin, 2008, pp. 31–46, DOI 10.1007/978-3-540-88403-3_3. MR2775645

[Ber08a] Daniel J Bernstein, *ChaCha, a variant of Salsa20*, Workshop record of sasc, 2008, pp. 3–5.

[Ber08b] Daniel J Bernstein, *The Salsa20 family of stream ciphers*, Lecture Notes in Computer Science **4986** (2008), 84–97.

[Ber11] Daniel J. Bernstein, *List decoding for binary Goppa codes*, Coding and cryptology, Lecture Notes in Comput. Sci., vol. 6639, Springer, Heidelberg, 2011, pp. 62–80, DOI 10.1007/978-3-642-20901-7_4. MR2834693

[BCLvV18] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal, *NTRU prime: reducing attack surface at low cost*, Selected areas in cryptography—SAC 2017, Lecture Notes in Comput. Sci., vol. 10719, Springer, Cham, 2018, pp. 235–260. MR3775587

[Ber11] Bertoni, Guido and Daemen, Joan and Peeters, Michael and Van Assche, Gilles, *The Keccak reference*, 2011.

[Bla03] Richard E Blahut, *Algebraic codes for data transmission*, Cambridge University Press, 2003.

[Bon99] Dan Boneh, *Twenty years of attacks on the RSA cryptosystem*, Notices Amer. Math. Soc. **46** (1999), no. 2, 203–213. MR1673760

[BSI18] BSI, *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*, Bundesamt für Sicherheit in der Informtionstechnik, 2018.

[CR12] Carlos Cid and Matt Robshaw, *The eSTREAM Portfolio in 2012*, ECRYPT II, 2012.

[CM13] Margaret Cozzens and Steven J. Miller, *The mathematics of encryption*, Mathematical World, vol. 29, American Mathematical Society, Providence, RI, 2013. An elementary introduction. MR3098499

[DR02] Joan Daemen and Vincent Rijmen, *The design of Rijndael*, Information Security and Cryptography, Springer-Verlag, Berlin, 2002. AES—the advanced encryption standard. MR1986943

[DCP08] Christophe De Canniere and Bart Preneel, *Trivium*, New Stream Cipher Designs, 2008, pp. 244–266.

[DR08] Tim Dierks and Eric Rescorla, *The Transport Layer Security* (*TLS*) *Protocol Version 1.2*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2008.

[DH76] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Trans. Information Theory **IT-22** (1976), no. 6, 644–654, DOI 10.1109/tit.1976.1055638. MR0437208

[Dwo01] Morris Dworkin, *Recommendation for Block Cipher Modes of Operation*, Technical Report SP 800-38A, National Institute of Standards and Technology, 2001.

[Dwo07] Morris J Dworkin, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode* (*GCM*) *and GMAC*, NIST Special Publication 800-38D, 2007.

[Dwo16] Morris J Dworkin, *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*, NIST Special Publication 800-38B, 2016.

[Edw74] H. M. Edwards, *Riemann's zeta function*, Academic Press [A subsidiary of Harcourt Brace Jovanovich, Publishers], New York-London, 1974. Pure and Applied Mathematics, Vol. 58. MR0466039

[Eke91] Artur K. Ekert, *Quantum cryptography based on Bell's theorem*, Phys. Rev. Lett. **67** (1991), no. 6, 661–663, DOI 10.1103/PhysRevLett.67.661. MR1118810

[FIP01] FIPS, *Federal Information Processing Standards Publication 197. Advanced Encryption Standard* (*AES*), 2001.

[FIP08] FIPS, *Federal Information Processing Standards Publication 198-1. The Keyed-Hash Message Authentication Code* (*HMAC*), 2008.

[FIP13] FIPS, *Federal Information Processing Standards Publication 186-4: Digital Signature Standard* (*DSS*), 2013.

[FIP15a] FIPS, *Federal Information Processing Standards Publication 180-4. Secure Hash Standard*, 2015.

[FIP15b] FIPS, *Federal Information Processing Standards Publication 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, 2015.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir, *Weaknesses in the key scheduling algorithm of RC4*, Selected areas in cryptography, Lecture Notes in Comput. Sci., vol. 2259, Springer, Berlin, 2001, pp. 1–24, DOI 10.1007/3-540-45537-X_1. MR2054424

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto, *Secure integration of asymmetric and symmetric encryption schemes*, J. Cryptology **26** (2013), no. 1, 80–101, DOI 10.1007/s00145-011-9114-1. MR3016824

[FOPS01] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern, *RSA-OAEP is secure under the RSA assumption*, Advances in cryptology—CRYPTO 2001 (Santa Barbara, CA), Lecture Notes in Comput. Sci., vol. 2139, Springer, Berlin, 2001, pp. 260–274, DOI 10.1007/3-540-44647-8_16. MR1931427

[Gal12] Steven D. Galbraith, *Mathematics of public key cryptography*, Cambridge University Press, Cambridge, 2012. MR2931758

[Gil16] Daniel Kahn Gillmor, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security* (*TLS*), Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2016.

[Gol01] Oded Goldreich, *Foundations of cryptography*, Cambridge University Press, Cambridge, 2001. Basic tools. MR1881185

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi, *Public-key cryptosystems from lattice reduction problems*, Advances in cryptology—CRYPTO '97 (Santa Barbara, CA, 1997), Lecture Notes in Comput. Sci., vol. 1294, Springer, Berlin, 1997, pp. 112–131, DOI 10.1007/BFb0052231. MR1630399

[GB08] Shafi Goldwasser and Mihir Bellare, *Lecture Notes on Cryptography*, Massachusetts Institute of Technology (MIT), 2008.

[Gro96] Lov K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing (Philadelphia, PA, 1996), ACM, New York, 1996, pp. 212–219, DOI 10.1145/237814.237866. MR1427516

[HPS$^+$17] Jeff Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang, *Choosing parameters for $NTRUEncrypt$*, Topics in cryptology—CT-RSA 2017, Lecture Notes in Comput. Sci., vol. 10159, Springer, Cham, 2017, pp. 3–18. MR3630855

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman, *NTRU: a ring-based public key cryptosystem*, Algorithmic number theory (Portland, OR, 1998), Lecture Notes in Comput. Sci., vol. 1423, Springer, Berlin, 1998, pp. 267–288, DOI 10.1007/BFb0054868. MR1726077

[HPS08] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman, *An introduction to mathematical cryptography*, Undergraduate Texts in Mathematics, Springer, New York, 2008. MR2433856

[HK07] Dennis Hofheinz and Eike Kiltz, *Secure hybrid encryption from weakened key encapsulation*, Advances in cryptology—CRYPTO 2007, Lecture Notes in Comput. Sci., vol. 4622, Springer, Berlin, 2007, pp. 553–571, DOI 10.1007/978-3-540-74143-5_31. MR2423870

[Hub96] Klaus Huber, *Note on decoding binary Goppa codes*, Electronics Letters **32** (1996), no. 2, 102–103.

[Hub03] Klaus Huber, *Taking pth roots modulo polynomials over finite fields*, Des. Codes Cryptogr. **28** (2003), no. 3, 303–311, DOI 10.1023/A:1024118322745. MR1976963

[HK97] Ran Canetti Hugo Krawczyk Mihir Bellare, *HMAC: Keyed-Hashing for Message Authentication*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 1997.

[IK03] Tetsu Iwata and Kaoru Kurosawa, *Stronger security bounds for OMAC, TMAC, and XCBC*, Progress in cryptology—INDOCRYPT 2003, Lecture Notes in Comput. Sci., vol. 2904, Springer, Berlin, 2003, pp. 402–415, DOI 10.1007/978-3-540-24582-7_30. MR2092397

[JOP14] Antoine Joux, Andrew Odlyzko, and Cécile Pierrot, *The past, evolving present, and future of the discrete logarithm*, Open problems in mathematics and computational science, Springer, Cham, 2014, pp. 5–36. MR3330876

[KL15] Jonathan Katz and Yehuda Lindell, *Introduction to modern cryptography*, 2nd ed., Chapman & Hall/CRC Cryptography and Network Security, CRC Press, Boca Raton, FL, 2015. MR3287369

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner, *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash*, NIST Special Publication 800-185, 2016.

[Ken05] Stephan T. Kent, *IP Encapsulating Security Payload* (*ESP*), Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2005.

[KM07] Neal Koblitz and Alfred J. Menezes, *Another look at "provable security"*, J. Cryptology **20** (2007), no. 1, 3–37, DOI 10.1007/s00145-005-0432-z. MR2340187

[KE10] Hugo Krawczyk and Pasi Eronen, *HMAC-based Extract-and-Expand Key Derivation Function* (*HKDF*), Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2010.

[Lam79] Leslie Lamport, *Constructing digital signatures from a one-way function*, Technical Report CSL-98, SRI International Palo Alto, 1979.

[LCM$^+$16] Adam Langley, WanTeh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson, *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security* (*TLS*), Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2016.

[LK08] Matt Lepinski and Stephen T. Kent, *Additional Diffie-Hellman Groups for Use with IETF Standards*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2008.

[LP11] Richard Lindner and Chris Peikert, *Better key sizes* (*and attacks*) *for LWE-based encryption*, Topics in cryptology—CT-RSA 2011, Lecture Notes in Comput. Sci., vol. 6558, Springer, Heidelberg, 2011, pp. 319–339, DOI 10.1007/978-3-642-19074-2_21. MR2804770

[LM10] Manfred Lochter and Johannes Merkle, *Elliptic Curve Cryptography* (*ECC*) *Brainpool Standard Curves and Curve Generation*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2010.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev, *On ideal lattices and learning with errors over rings*, J. ACM **60** (2013), no. 6, Art. 43, 35, DOI 10.1145/2535925. MR3144913

[MV04] David A. McGrew and John Viega, *The security and performance of the Galois/counter mode* (*GCM*) *of operation*, Progress in cryptology—INDOCRYPT 2004, Lecture Notes in Comput. Sci., vol. 3348, Springer, Berlin, 2004, pp. 343–355, DOI 10.1007/978-3-540-30556-9_27. MR2148337

[MOV93] Alfred J. Menezes, Tatsuaki Okamoto, and Scott A. Vanstone, *Reducing elliptic curve logarithms to logarithms in a finite field*, IEEE Trans. Inform. Theory **39** (1993), no. 5, 1639–1646, DOI 10.1109/18.259647. MR1281712

[MvOV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press Series on Discrete Mathematics and its Applications, CRC Press, Boca Raton, FL, 1997. With a foreword by Ronald L. Rivest. MR1412797

[MR09] Daniele Micciancio and Oded Regev, *Lattice-based cryptography*, Post-quantum cryptography, Springer, Berlin, 2009, pp. 147–191, DOI 10.1007/978-3-540-88702-7_5. MR2590647

[MKJR16] Kathleen M. Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch, *PKCS #1: RSA Cryptography Specifications Version 2.2*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2016.

[NYHR05] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn, *The Kerberos Network Authentication Service* (*V5*), Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2005.

[NC12] Robert Niebuhr and Pierre-Louis Cayrel, *Broadcast attacks against code-based schemes*, Research in cryptology, Lecture Notes in Comput. Sci., vol. 7242, Springer, Heidelberg, 2012, pp. 1–17, DOI 10.1007/978-3-642-34159-5_1. MR3021183

[NC00] Michael A. Nielsen and Isaac L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, Cambridge, 2000. MR1796805

[NL18] Yoav Nir and Adam Langley, *ChaCha20 and Poly1305 for IETF Protocols*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2018.

[OP01] Tatsuaki Okamoto and David Pointcheval, *The gap-problems: a new class of problems for the security of cryptographic schemes*, Public key cryptography (Cheju Island, 2001), Lecture Notes in Comput. Sci., vol. 1992, Springer, Berlin, 2001, pp. 104–118, DOI 10.1007/3-540-44586-2_8. MR1898028

[OS09] Raphael Overbeck and Nicolas Sendrier, *Code-based cryptography*, Post-quantum cryptography, Springer, Berlin, 2009, pp. 95–145, DOI 10.1007/978-3-540-88702-7_4. MR2590646

[PP10] Christof Paar and Jan Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer, 2010.

[Pat75] N. J. Patterson, *The algebraic decoding of Goppa codes*, IEEE Trans. Information Theory **IT-21** (1975), 203–207, DOI 10.1109/tit.1975.1055350. MR0379009

[PM07] Goutam Paul and Subhamoy Maitra, *Permutation after RC4 key scheduling reveals the secret key*, International workshop on selected areas in cryptography, 2007, pp. 360–377.

[Pei14] Chris Peikert, *A decade of lattice cryptography*, Found. Trends Theor. Comput. Sci. **10** (2014), no. 4, i—iii, 283–424, DOI 10.1561/0400000074. MR3494162

[Poi00] David Pointcheval, *Chosen-ciphertext security for any one-way cryptosystem*, Public key cryptography (Melbourne, 2000), Lecture Notes in Comput. Sci., vol. 1751, Springer, Berlin, 2000, pp. 129–146, DOI 10.1007/978-3-540-46588-1_10. MR1864776

[Pom96] Carl Pomerance, *A tale of two sieves*, Notices Amer. Math. Soc. **43** (1996), no. 12, 1473–1485. MR1416721

[Reg09] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, J. ACM **56** (2009), no. 6, Art. 34, 40, DOI 10.1145/1568318.1568324. MR2572935

[Res18] Eric Rescorla, *The Transport Layer Security* (*TLS*) *Protocol Version 1.3*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2018.

[RP11] Eleanor Rieffel and Wolfgang Polak, *Quantum computing*, Scientific and Engineering Computation, MIT Press, Cambridge, MA, 2011. A gentle introduction. MR2791092

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM **21** (1978), no. 2, 120–126, DOI 10.1145/359340.359342. MR700103

[RB08] Matthew Robshaw and Olivier Billet, *New stream cipher designs*, LNCS, vol. 4986, Springer, 2008.

[Ros12] Kenneth H Rosen, *Discrete Mathematics and its Applications*, 7th ed., McGraw-Hill, 2012.

[Rot06] Ron Roth, *Introduction to Coding Theory*, Cambridge University Press, 2006.

[RSN$^+$10] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, Technical Report SP 800-22 Revision 1a, National Institute of Standards of Technology, 2010.

[Sag18] Sage Developers, *Sagemath, the Sage Mathematics Software System* (*Version 8.5*), https://www.sagemath.org, 2018.

[Sha49] C. E. Shannon, *Communication theory of secrecy systems*, Bell System Tech. J. **28** (1949), 656–715, DOI 10.1002/j.1538-7305.1949.tb00928.x. MR0032133

[She17] Thomas R. Shemanske, *Modern cryptography and elliptic curves*, Student Mathematical Library, vol. 83, American Mathematical Society, Providence, RI, 2017. A beginner's guide. MR3676088

[Sho94] Peter W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, 35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994), IEEE Comput. Soc. Press, Los Alamitos, CA, 1994, pp. 124–134, DOI 10.1109/SFCS.1994.365700. MR1489242

[Sho09] Victor Shoup, *A computational introduction to number theory and algebra*, 2nd ed., Cambridge University Press, Cambridge, 2009. MR2488898

[Sil09] Joseph H. Silverman, *The arithmetic of elliptic curves*, 2nd ed., Graduate Texts in Mathematics, vol. 106, Springer, Dordrecht, 2009. MR2514094

[SPL06] Jun H. Song, Radha Poovendran, and Jicheol Lee, *The AES-CMAC-96 Algorithm and Its Use with IPsec*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2006.

[SPLI06] Jun H. Song, Radha Poovendran, Jicheol Lee, and Tetsu Iwata, *The AES-CMAC Algorithm*, Internet Request for Comments, RFC Editor, Fremont, CA, USA, 2006.

[SBK$^+$17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov, *The first collision for full SHA-1*, Advances in cryptology—CRYPTO 2017. Part I, Lecture Notes in Comput. Sci., vol. 10401, Springer, Cham, 2017, pp. 570–596. MR3703211

[Was08] Lawrence C Washington, *Elliptic curves: Number Theory and Cryptography*, CRC Press, 2008.

[TW06] Wade Trappe and Lawrence C. Washington, *Introduction to cryptography with coding theory*, 2nd ed., Pearson Prentice Hall, Upper Saddle River, NJ, 2006. MR2372272

[WJW$^+$14] Klaus Weltner, Sebastian John, Wolfgang Weber, et al., *Mathematics for Physicists and Engineers: Fundamentals and Interactive Study Guide*, Springer, 2014.

[Zen07] Erik Zenner, *Why IV Setup for Stream Ciphers is Difficult*, Dagstuhl seminar proceedings, 2007.

Index

PUBLISHED TITLES IN THIS SERIES

40 **Heiko Knospe,** A Course in Cryptography, 2019
36 **Cesar E. Silva,** Invitation to Real Analysis, 2019
35 **Álvaro Lozano-Robledo,** Number Theory and Geometry, 2019
34 **C. Herbert Clemens,** Two-Dimensional Geometries, 2019
33 **Brad G. Osgood,** Lectures on the Fourier Transform and Its Applications, 2019
32 **John M. Erdman,** A Problems Based Course in Advanced Calculus, 2018
31 **Benjamin Hutz,** An Experimental Introduction to Number Theory, 2018
30 **Steven J. Miller,** Mathematics of Optimization: How to do Things Faster, 2017
29 **Tom L. Lindstrøm,** Spaces, 2017
28 **Randall Pruim,** Foundations and Applications of Statistics: An Introduction Using R, Second Edition, 2018
27 **Shahriar Shahriari,** Algebra in Action, 2017
26 **Tamara J. Lakins,** The Tools of Mathematical Reasoning, 2016
25 **Hossein Hosseini Giv,** Mathematical Analysis and Its Inherent Nature, 2016
24 **Helene Shapiro,** Linear Algebra and Matrices, 2015
23 **Sergei Ovchinnikov,** Number Systems, 2015
22 **Hugh L. Montgomery,** Early Fourier Analysis, 2014
21 **John M. Lee,** Axiomatic Geometry, 2013
20 **Paul J. Sally, Jr.,** Fundamentals of Mathematical Analysis, 2013
19 **R. Clark Robinson,** An Introduction to Dynamical Systems: Continuous and Discrete, Second Edition, 2012
18 **Joseph L. Taylor,** Foundations of Analysis, 2012
17 **Peter Duren,** Invitation to Classical Analysis, 2012
16 **Joseph L. Taylor,** Complex Variables, 2011
15 **Mark A. Pinsky,** Partial Differential Equations and Boundary-Value Problems with Applications, Third Edition, 1998
14 **Michael E. Taylor,** Introduction to Differential Equations, 2011
13 **Randall Pruim,** Foundations and Applications of Statistics, 2011
12 **John P. D'Angelo,** An Introduction to Complex Analysis and Geometry, 2010
11 **Mark R. Sepanski,** Algebra, 2010
10 **Sue E. Goodman,** Beginning Topology, 2005
9 **Ronald Solomon,** Abstract Algebra, 2003
8 **I. Martin Isaacs,** Geometry for College Students, 2001
7 **Victor Goodman and Joseph Stampfli,** The Mathematics of Finance, 2001
6 **Michael A. Bean,** Probability: The Science of Uncertainty, 2001
5 **Patrick M. Fitzpatrick,** Advanced Calculus, Second Edition, 2006
4 **Gerald B. Folland,** Fourier Analysis and Its Applications, 1992
3 **Bettina Richmond and Thomas Richmond,** A Discrete Transition to Advanced Mathematics, 2004
2 **David Kincaid and Ward Cheney,** Numerical Analysis: Mathematics of Scientific Computing, Third Edition, 2002
1 **Edward D. Gaughan,** Introduction to Analysis, Fifth Edition, 1998

This book provides a compact course in modern cryptography. The mathematical foundations in algebra, number theory and probability are presented with a focus on their cryptographic applications. The text provides rigorous definitions and follows the provable security approach. The most relevant cryptographic schemes are covered, including block ciphers, stream ciphers, hash functions, message authentication codes, public-key encryption, key establishment, digital signatures and elliptic curves. The current developments in post-quantum cryptography are also explored, with separate chapters on quantum computing, lattice-based and code-based cryptosystems.

Many examples, figures and exercises, as well as SageMath (Python) computer code, help the reader to understand the concepts and applications of modern cryptography. A special focus is on algebraic structures, which are used in many cryptographic constructions and also in post-quantum systems. The essential mathematics and the modern approach to cryptography and security prepare the reader for more advanced studies.

The text requires only a first-year course in mathematics (calculus and linear algebra) and is also accessible to computer scientists and engineers. This book is suitable as a textbook for undergraduate and graduate courses in cryptography as well as for self-study.

ISBN 978-1-4704-5055-7

9 781470 450557

AMSTEXT/40

For additional information and updates on this book, visit
www.ams.org/bookpages/amstext-40

AMS
www.ams.org

The Sally SERIES

This series was founded by the highly respected mathematician and educator, Paul J. Sally, Jr.